STC12C2052AD series MCU
STC12LE2052AD series MCU
Data Sheet

STC MCU Limited

Update date: 2011-7-15

CONTENTS

Chapter 1. Introduction

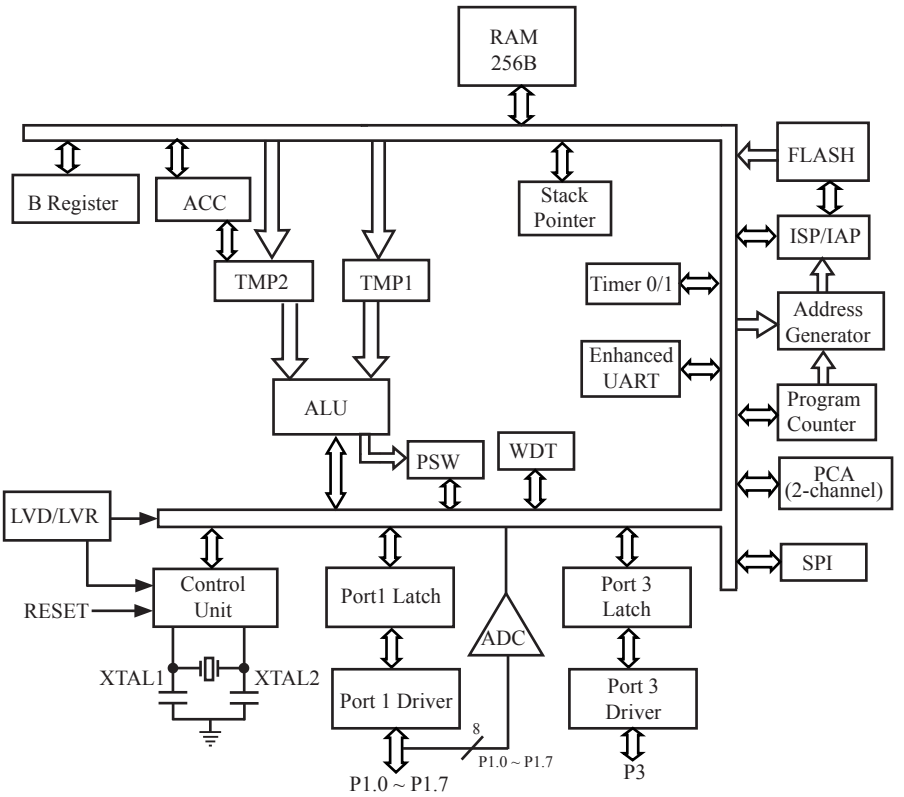
STC12C2052AD is a single-chip microcontroller based on a high performance 1T architecture 8051 CPU, which is produced by STC MCU Limited. With the enhanced kernel, STC12C2052AD executes instructions in 1~6 clock cycles (about 8~12 times the rate of a standard 8051 device), and has a fully compatible instruction set with industrial-standard 8051 series microcontroller. In-System-Programming (ISP) and In-Application-Programming (IAP) support the users to upgrade the program and data in system. ISP allows the user to download new code without removing the microcontroller from the actual end product; IAP means that the device can write non-valatile data in Flash memory while the application program is running. The STC12C2052AD retains all features of the standard 80C51. In addition, the STC12C2052AD has a 9-sources, 4-priority-level interrupt structure, 8-bit ADC (100 thousands times per second), on-chip crystal oscillator, MAX810 special reset circuit, 2-channel PCA and PWM, SPI, a one-time enabled Watchdog Timer and so on.

1.1 Features

- Enhanced 8051 Central Processing Unit ,1T per machine cycle, faster 8~12 times than the rate of a standard 8051.
- Operating voltage range: 5.5V ~ 3.5V or 2.2V ~ 3.6V (STC12LE2052AD).
- Operating frequency range: 0- 35MHz, is equivalent to standard 8051:0~420MHz
- On-chip 1K/2K/3K/4K/5K... FLASH program memory with flexible ISP/IAP capability
- On-chip 256 byte RAM
- Power control: idle mode(can be waked up by any interrupt) and power-down mode(can be waked up by external interrupts).
- Code protection for flash memory access
- Excellent noise immunity, very low power consumption
- Four 16-bit timer/counter, be compatible with Timer0/Timer1 of standard 8051, 2-channel PCA can be available as two timers.
- 9 vector-address, 4 level priority interrupt capability
- One enhanced UART with hardware address-recognition and frame-error detection function
- One 15 bits Watch-Dog-Timer with 8-bit pre-scaler (one-time-enabled)
- SPI Master/Slave communication interface
- Two channel Programmable Counter Array (PCA)
- 8-bit, 8-channel high-speed Analog-to-Digital Converter (ADC), up to 100 thousands times per second
- Simple internal RC oscillator and external crystal clock
- Three power management modes: idle mode, slow down mode and power-down mode
- Power down mode can be woken-up by P3.2/INT0, P3.3/INT1, P3.4/T0, P3.5/T1, P3.0/RxD, P3.7/PCA0, and P3.5/PCA1
- Operation Temperature: -40 ~ + 85℃ (industrial) / 0 ~ 75℃ (Commercial)
- 15 common programmable I/O ports are available
- Programmable clock output Function. T0 output the clock on P1.0, T1 output the clock on P1.1.
- Five package type : SOP-20, DIP-20, LSSOP-20.

1.2 Block diagram

The CPU kernel of STC12C2052AD is fully compatible to the standard 8051 microcontroller, maintains all instruction mnemonics and binary compatibility. With some great architecture enhancements, STC12C2052AD executes the fastest instructions per clock cycle. Improvement of individual programs depends on the actual instructions used.

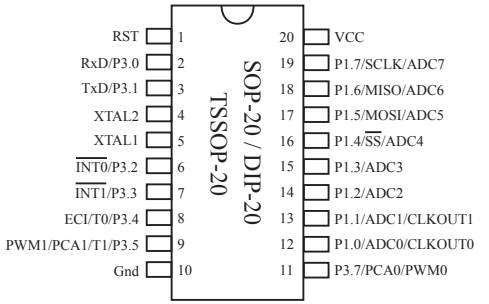


STC12C2052AD Block Diagram

1.3 Pin Configurations

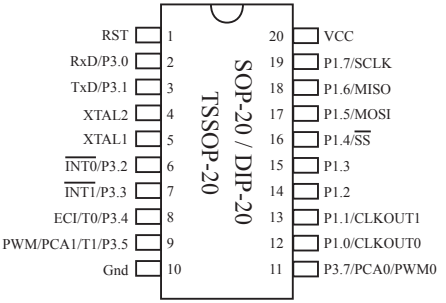
All packages meet the European Union RoHS standards. LQFP-32 also conform to the Green standard.

Packages such as SOP-20 are strongly recommended though the traditional DIP packages are steady supplied.



STC12C2052AD series (with A/D Converter)，20-Pin

15 I/O ports



STC12C2052 series (without A/D Converter)，20-Pin

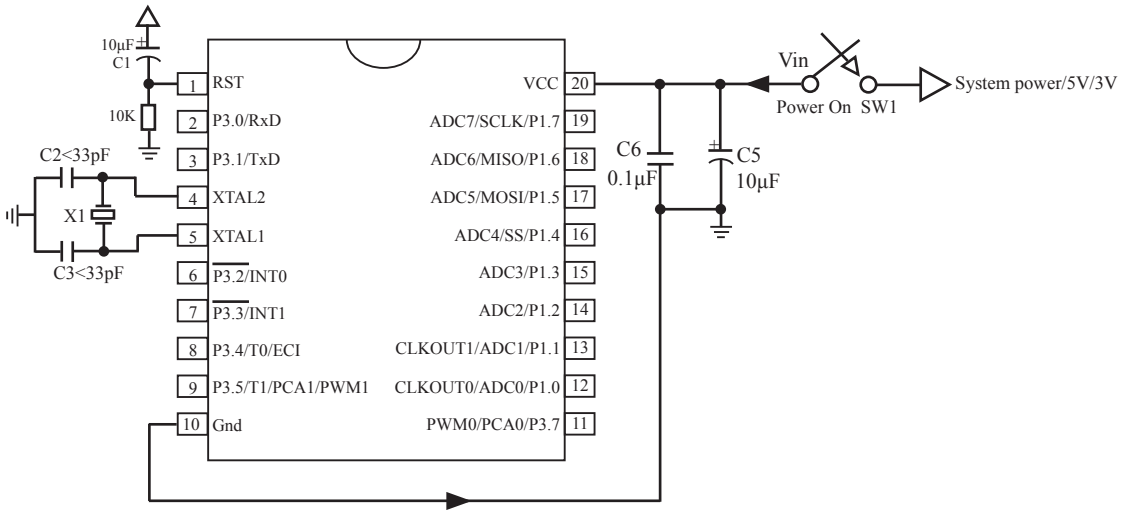
Super small package： TSSOP-20, 6.4mm×6.4mm

1.4 STC12C2052AD series Selection Table

Type 1T 8051 MCU	Operation Voltage (V)	Flash (Byte)	SRAM (Byte)	Timer T0&T1	PCA Timer	Programmable Clock Output	UART	EEP ROM	16-bit PCA/ 8-bit PWM D/A	A/D 8-ch	WDT	Built-in Reset	SPI	Package of 20-Pin (15 I/O ports)
STC12C2052AD series Selection Table														
STC12C1052	5.5-3.5	1K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	SOP/LSSOP/DIP
STC12C1052AD	5.5-3.5	1K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	SOP/LSSOP/DIP
STC12C2052	5.5-3.5	2K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	SOP/LSSOP/DIP
STC12C2052AD	5.5-3.5	2K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	SOP/LSSOP/DIP
STC12C3052	5.5-3.5	3K	256	Y	2	Y	Y	IAP	2-ch		Y	Y	Y	SOP/LSSOP/DIP
STC12C3052AD	5.5-3.5	3K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	SOP/LSSOP/DIP
STC12C4052	5.5-3.5	4K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	SOP/LSSOP/DIP
STC12C4052AD	5.5-3.5	4K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	SOP/LSSOP/DIP
STC12C5052	5.5-3.5	5K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	Application program can be modified in application program area (AP area)
STC12C5052AD	5.5-3.5	6K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	
STC12LE2052AD series Selection Table														
STC12LE1052	3.6-2.2	1K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	SOP/LSSOP/DIP
STC12LE1052AD	3.6-2.2	1K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	SOP/LSSOP/DIP
STC12LE2052	3.6-2.2	2K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	SOP/LSSOP/DIP
STC12LE2052AD	3.6-2.2	2K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	SOP/LSSOP/DIP
STC12LE3052	3.6-2.2	3K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	SOP/LSSOP/DIP
STC12LE3052AD	3.6-2.2	3K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	SOP/LSSOP/DIP
STC12LE4052	3.6-2.2	4K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	SOP/LSSOP/DIP
STC12LE4052AD	3.6-2.2	4K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	SOP/LSSOP/DIP
STC12LE5052	3.6-2.2	5K	256	Y	2	Y	Y	Y	2-ch		Y	Y	Y	Application program can be modified in application program area (AP area)
STC12LE5052AD	3.6-2.2	5K	256	Y	2	Y	Y	Y	2-ch	8-bit	Y	Y	Y	

1.5 STC12C2052AD series Minimum Application System

When the frequency of crystal oscillator is lower than 12MHz,
it is suggested not to use C1 and R1 replaced by 1K
resistor connect to ground.
But R/C reset circuit is also suggest to reserve.



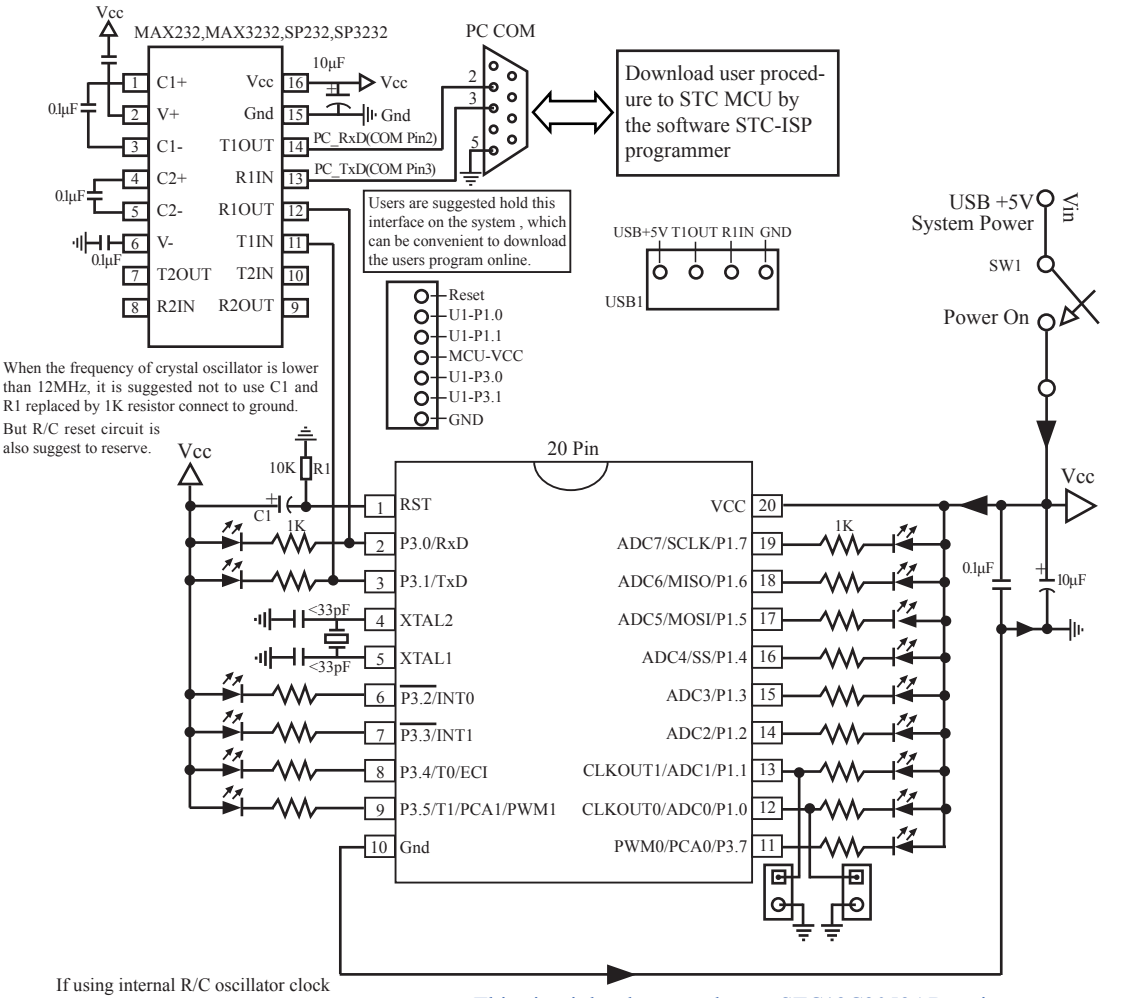
About crystals circuit:

If using internal R/C oscillator clock (4MHz ~ 8MHz, manufacturing error), XTAL1 and XTAL2 pin should be floated.

If External clock frequency is higher than 33MHz, it is recommended to directly use external active crystals which clock are input from XTAL1 pin and XTAL2 pin must be floated.

1.6 STC12C2052AD series MCU Typical Application Circuit for ISP

—— MCU should be connected to computer through RS-232 converter to download program



If using internal R/C oscillator clock (4MHz ~ 8MHz, manufacturing error), XTAL1 and XTAL2 pin should be floated.

If External clock frequency is higher than 33MHz, it is recommended to directly use external active crystals which clock are input from XTAL1 pin and XTAL2 pin must be floated.

This circuit has been made as a STC12C2052AD series microcontroller ISP download programming tool

Users in their target system, such as the P3.0/P3.1 through the RS-232 level shifter connected to the computer after the conversion of ordinary RS-232 serial port to connect the system programming / upgrading client software. If the user panel recommended no RS-232 level converter, should lead to a socket, with Gnd/P3.1/P3.0/Vcc four signal lines, so that the user system can be programmed directly. Of course, if the six signal lines can lead to Gnd/P3.1/P3.0/Vcc/P1.1/P1.0 as well, because you can download the program by P1.0/P1.1 ISP ban. If you can Gnd/P3.1/P3.0/Vcc/P1.1/P1.0/Reset seven signal lines leads to better, so you can easily use "offline download board (no computer)" .

ISP programming on the Theory and Application Guide to see "STC12C2052AD Series MCU Development / Programming Tools Help"section. In addition, we have standardized programming download tool, the user can then program into the goal in the above systems, you can borrow on top of it RS-232 level shifter connected to the computer to download the program used to do. Programming a chip roughly be a few seconds, faster than the ordinary universal programmer much faster, there is no need to buy expensive third-party programmer?.

PC STC-ISP software downloaded from the website

1.7 Pin Descriptions

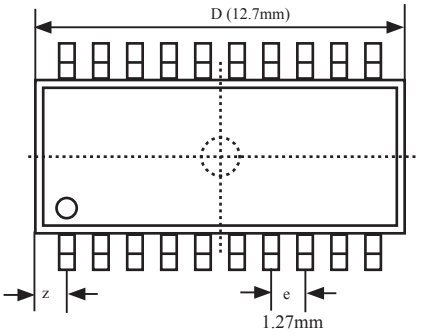
MNEMONIC	Pin Number				Description	
	LQFP-32	SOP-32	SOP-28/ SKDIP-28	SOP-20/DIP-20/ TSSOP-20		
P1.0/ADC0/CLKOUT0	16	20	18	12	P1.0	common I/O port PORT1[0]
					ADC0	Analog to Digital Converter Input channel 0
					CLKOUT0	Programmable clock output of Timer/ couter 0
P1.1/ADC1/CLKOUT1	17	21	19	13	P1.1	common I/O port PORT1[1]
					ADC1	Analog to Digital Converter Input channel 1
					CLKOUT1	rogrammable clock output of Timer/ counter 1
P1.2/ADC2	18	22	20	14	P1.2	common I/O port PORT1[2]
					ADC2	Analog to Digital Converter Input channel 2
P1.3/ADC3	20	24	21	15	P1.3	common I/O port PORT1[3]
					ADC3	Analog to Digital Converter Input channel 3
P1.4/ADC4/$\overline{SS}$	21	25	22	16	P1.4	common I/O port PORT1[4]
					ADC4	Analog to Digital Converter Input channel 4
					$\overline{SS}$	slave-select signal of serial peripheral interface, which is active low.
P1.5/ADC5/MOSI	23	27	23	17	P1.5	common I/O port PORT1[5]
					ADC5	Analog to Digital Converter Input channel 5
					MOSI	Master Out, Slave In signal
P1.6/ADC6/MISO	24	28	24	18	P1.6	common I/O port PORT1[6]
					ADC5	Analog to Digital Converter Input channel 6
					MISO	Master In, Slave Out signal
P1.7/ADC7/SCLK	25	29	25	19	P1.7	common I/O port PORT1[7]
					ADC7	Analog to Digital Converter Input channel 7
					SCLK	The clock signal of serial peripheral interface
P3.0/RxD	32	4	4	2	P3.0	common I/O port PORT3[0]
					RxD	Serial recive port
P3.1/TxD	1	5	5	3	P3.1	common I/O port PORT3[1]
					TxD	Serial transmit port

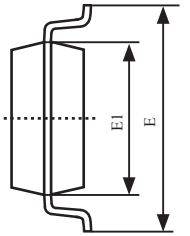
MNEMONIC	Pin Number				Description	
	LQFP-32	SOP-32	SOP-28/ SKDIP-28	SOP-20/DIP-20/ TSSOP-20		
P3.2/$\overline{\text{INT0}}$	5	9	8	6	P3.2	common I/O port PORT3[2]
					$\overline{\text{INT0}}$	External interrupt 0
P3.3/$\overline{\text{INT1}}$	7	11	9	7	P3.3	common I/O port PORT3[3]
					$\overline{\text{INT1}}$	External interrupt 1
P3.4/T0/ECI	8	12	10	8	P3.4	common I/O port PORT3[4]
					T0	Timer/counter 0 input
					ECI	PCA count input
P3.5/T1/PCA1/PWM1	9	13	11	9	P3.5	common I/O port PORT3[5]
					T1	Timer/counter 1 input
					PCA1	PCA module 1 Capture input
					PWM1	PWM module 1 output
P3.7/PCA0/PWM0	15	19	17	11	P3.7	common I/O port PORT3[7]
					PCA0	PCA module 0 Capture input
					PWM0	PWM module 0 output
RST	31	3	3	1	Reset input	
XTAL1	4	8	7	5	Input to the inverting oscillator amplifier. Receives the external oscillator signal when an external oscillator is used.	
XTAL2	3	7	6	4	Output from the inverting amplifier. This pin should be floated when an external oscillator is used.	
VCC	28	32	28	20	Power	
Gnd	12	16	14	10	circuit ground potential	

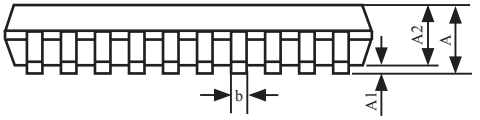
1.8 Package Dimension Drawings

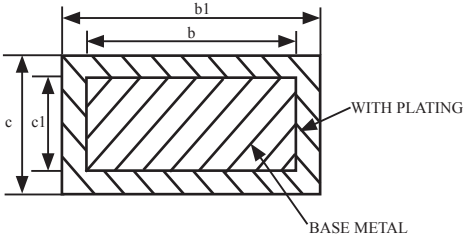
20-Pin Small Outline Package (SOP-20)

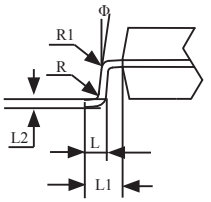
Dimensions in Inches and (Millimeters)







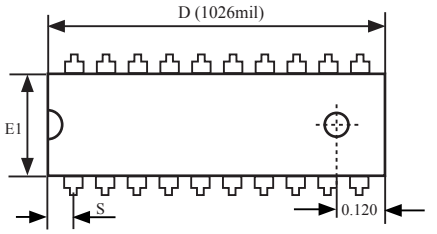


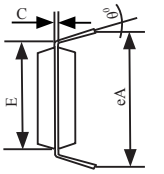


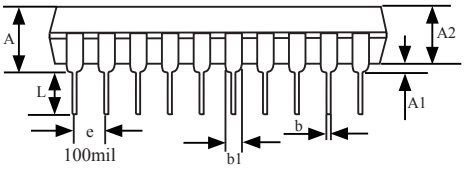
COMMON DIMENSIONS			
(UNITS OF MEASURE = MILLMETER)			
SYMBOL	MIN	NOM	MAX
A	2.465	2.515	2.565
A1	0.100	0.150	0.200
A2	2.100	2.300	2.500
b1	0.366	0.426	0.486
b	0.356	0.406	0.456
c	0.234	-	0.274
c1	0.224	0.254	0.274
D	12.500	12.700	12.900
E	10.206	10.306	10.406
E1	7.450	7.500	7.550
e		1.270	
L	0.800	0.864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

20-Pin Plastic Dual Inline Package (DIP-20)

Dimensions in Inches



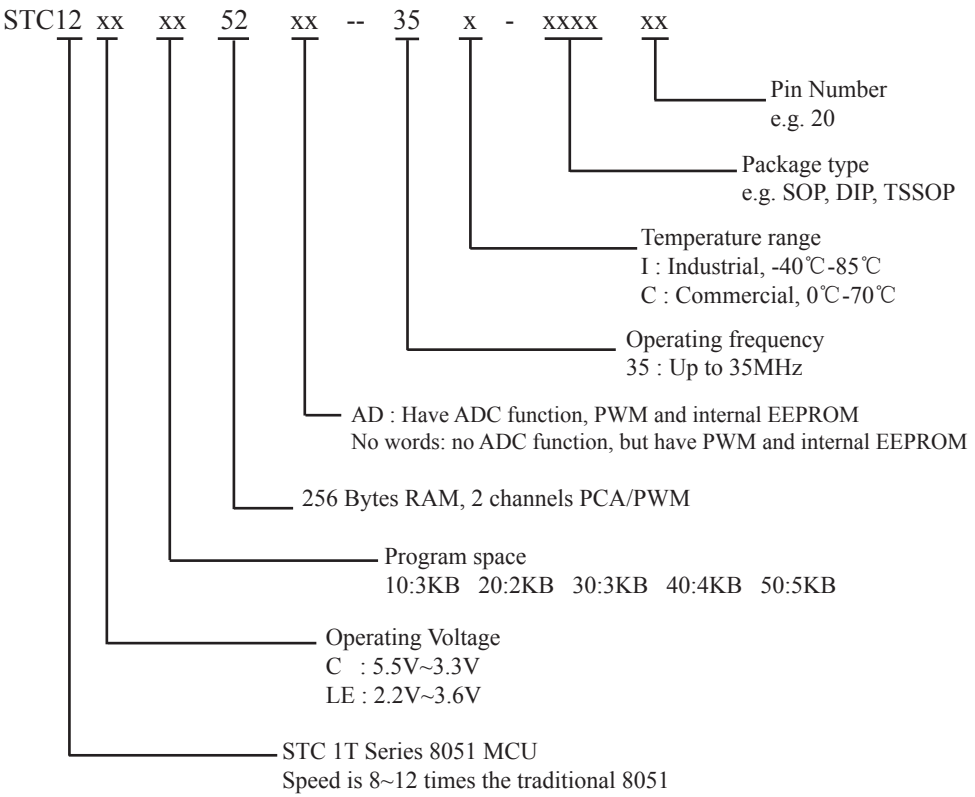




COMMON DIMENSIONS			
(UNITS OF MEASURE = INCH)			
SYMBOL	MIN	NOM	MAX
A	-	-	0.175
A1	0.015	-	-
A2	0.125	0.13	0.135
b	0.016	0.018	0.020
b1	0.058	0.060	0.064
C	0.008	0.010	0.11
D	1.012	1.026	1.040
E	0.290	0.300	0.310
E1	0.245	0.250	0.255
e	0.090	0.100	0.110
L	0.120	0.130	0.140
θ^0	0	-	15
eA	0.355	0.355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

UNIT: INCH 1 inch = 1000 mil

1.9 STC12C2052AD series MCU naming rules



1.10 Global Unique Identification Number (ID)

STC 1T MCU 12C2052AD series, each MCU has a unique identification number (ID). User can use “MOV @Ri” instruction read RAM unit F1~F7 to get the ID number after power on. If users need to the unique identification number to encrypt their procedures, detecting the procedures not be illegally modified should be done first.

//The following example program written by C language is to read internal ID number

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* If you want to use the program or the program referenced in the  --*/
/* article, please specify in which data and procedures from STC    --*/
/*---------------------------------------------------------------------------*/
#include<reg51.h>
#include<intrins.h>
sfr         ISP_CONTR         =  0xE7;

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

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

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

            unsigned char idata *idata_point;
```

```
	serial_port_initial();			//initialize serial port
//	display_MCU_Start_Led();		//MCU begin to run when LED is be lighted
//	send_UART(0x34);
//	send_UART(0xa7);

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

	while(1);
}

void serial_port_initial()
{
	SCON	=	0x50;			//0101,0000  8-bit variable baud rate，No parity
	TMOD	=	0x21;			//0011,0001  Timer1 as 8-bit auto-reload Timer
	TH1	=	RELOAD_COUNT;		//Set the auto-reload parameter
	TL1	=	RELOAD_COUNT;
	TR1	=	1;
	ES	=	1;
	EA	=	1;
}

void send_UART(unsigned char i)
{
	ES	=	0;
	TI	=	0;
	SBUF	=	i;
	while(TI ==0);
	TI	=	0;
	ES	=	1;
}

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

```
                if(k==Self_Define_ISP_Download_Command)          //Self-define download command
                {
                        delay();                                 //just delay 1 second
                        delay();
                        soft_reset_to_ISP_Monitor();             //Soft rese to ISP Monitor
                }
                send_UART(k);
        }
        else
        {
                TI = 0;
        }
}


void soft_reset_to_ISP_Monitor(void)
{
        ISP_CONTR = 0x60;                                //0110,0000   Soft rese to 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;
                delay();
                MCU_Start_Led = 1;
                delay();
                MCU_Start_Led = 0;
        }
}
```

Chapter 2. Clock, Power Management and Reset

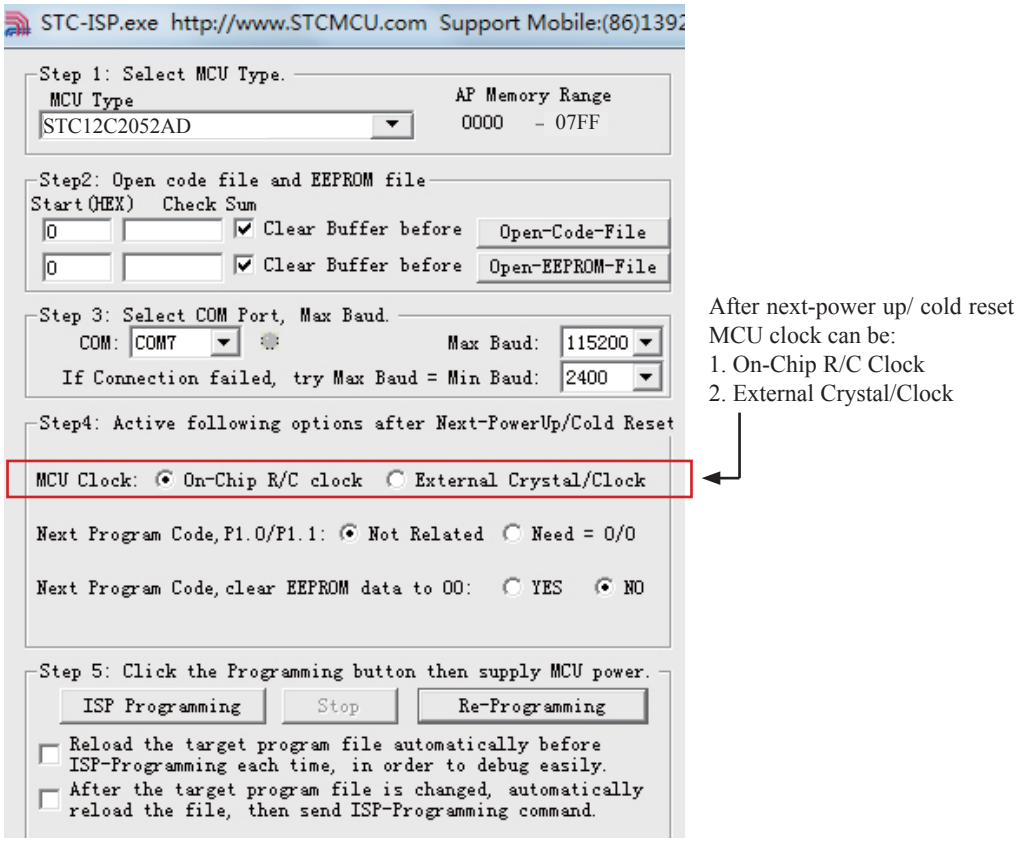
2.1 Clock

2.1.1 On-Chip R/C Clock and External Crystal/Clcok are Optional in STC-ISP.exe

STC12C2052AD series is STC 1T MCU whose system clock is compatible with traditional 8051 MCU.

There are two clock sources available for STC12C2052AD. One is the clock from crystal oscillator and the other is from internal simple RC oscillator. The internal built-in RC oscillator can be used to replace the external crystal oscillator in the application which doesn't need an exact system clock. On-chip R/C clock is selected first in STC-ISP Writer/Programmer because the manufacturer's selection is on-chip R/C clock.

To enable the external crystal oscillator, user should enable the option "External Crystal/Clock" by STC-ISP Writer/Programmer.



After next-power up/ cold reset
MCU clock can be:
1. On-Chip R/C Clock
2. External Crystal/Clock

2.1.2 Divider for System Clock

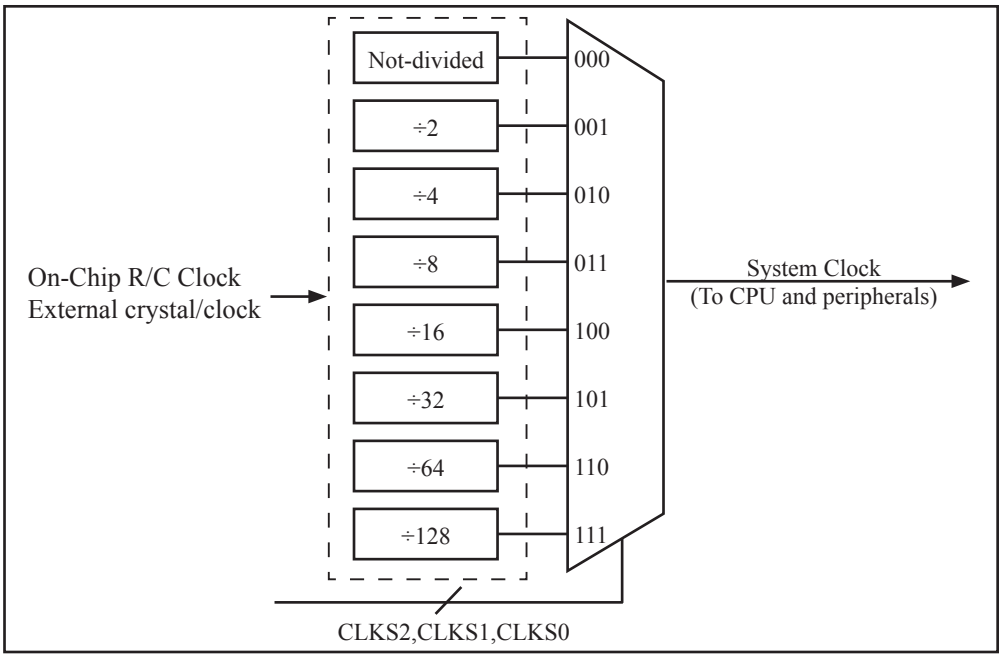
A clock divider(CLK_DIV) is designed to slow down the operation speed of STC12C2052AD, to save the operating power dynamically. User can slow down the MCU by means of writing a non-zero value to the CLKS[2:0] bits in the CLK_DIV register. This feature is especially useful to save power consumption in idle mode as long as the user changes the CLKS[2:0] to a non-zero value before entering the idle mode.

CLK_DIV register (Clock Divider)

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

B2-B0 (CLKS2-CLKS0) :

000 External crystal/clock or On-Chip R/C clock is not divided (default state)
001 External crystal/clock or On-Chip R/C clock is divided by 2.
010 External crystal/clock or On-Chip R/C clock is divided by 4.
011 External crystal/clock or On-Chip R/C clock is divided by 8.
100 External crystal/clock or On-Chip R/C clock is divided by 16.
101 External crystal/clock or On-Chip R/C clock is divided by 32.
110 External crystal/clock or On-Chip R/C clock is divided by 64.
111 External crystal/clock or On-Chip R/C clock is divided by 128.



Clock Structure

2.1.3 How to Know Internal RC Oscillator frequency(Internal clock frequency)

STC 1T MCU 12C2052AD series in addition to traditional external clock, but also the option of using the internal RC oscillator clock source. If select internal RC oscillator, external crystal can be saved. XTAL1 and XTAL2 floating. Relatively large errors due to internal clock, so high requirements on the timing or circumstances have serial communication is not recommended to use the internal oscillator. User can use “MOV @Ri” instruction read RAM unit FC~FF to get the internal oscillator frequency of the factory and read RAM unit F8~FB to get internal oscillator frequency of last used to download programs within the internal oscillator after power on.

//The following example program written by C language is to read internal R/C clock frequency

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* If you want to use the program or the program referenced in the  --*/
/* article, please specify in which data and procedures from STC    --*/
/*---------------------------------------------------------------------------*/
#include<reg51.h>
#include<intrins.h>
sfr         ISP_CONTR       =  0xE7;

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

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

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

          unsigned char idata *idata_point;
```

```
        serial_port_initial();                  //initialize serial port
//      display_MCU_Start_Led();                //MCU begin to run when LED is be lighted
//      send_UART(0x34);
//      send_UART(0xa7);

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

        while(1);
}

void serial_port_initial()
{
        SCON    =   0x50;                       //0101,0000  8-bit variable baud rate，No parity
        TMOD    =   0x21;                       //0011,0001  Timer1 as 8-bit auto-reload Timer
        TH1     =   RELOAD_COUNT;               //Set the auto-reload parameter
        TL1     =   RELOAD_COUNT;
        TR1     =   1;
        ES      =   1;
        EA      =   1;
}

void send_UART(unsigned char i)
{
        ES      =   0;
        TI      =   0;
        SBUF    =   i;
        while(TI ==0);
        TI      =   0;
        ES      =   1;
}

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

```
                if(k==Self_Define_ISP_Download_Command)          //Self-define download command
                {
                        delay();                                 //just delay 1 second
                        delay();
                        soft_reset_to_ISP_Monitor();             //Soft rese to ISP Monitor
                }
                send_UART(k);
        }
        else
        {
                TI = 0;
        }
}

void soft_reset_to_ISP_Monitor(void)
{
        ISP_CONTR = 0x60;                                //0110,0000  Soft rese to 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;
                delay();
                MCU_Start_Led = 1;
                delay();
                MCU_Start_Led = 0;
        }
}
```

2.1.4 Programmable Clock Output

STC12C2052AD series MCU have two channel programmable clock outputs, they are Timer 0 programmable clock output CLKOUT0(P1.0) and Timer 1 programmable clock output CLKOUT1(P1.1).

There are some SFRs about programmable clock output as shown below.

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
AUXR	Auxiliary register	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
WAKE_CLKO	CLK_Output Power down Wake-up control register	8FH	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO	0000 xx00B

The satement (used in C language) of Special function registers AUXR/WAKE_CLKO:

```
sfr     AUXR      =  0x8E;     //The address statement of Special function register AUXR
sfr     WAKE_CLKO =  0x8F;     //The address statement of SFR WAKE_CLKO
```

The satement (used in Assembly language) of Special function registers AUXR/WAKE_CLKO:

```
AUXR      EQU  0x8E      ;The address statement of Special function register AUXR
WAKE_CLKO EQU  0x8F      ;The address statement of SFR WAKE_CLKO
```

1. AUXR: Auxiliary register (Non bit-addressable)

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

T0x12 : Timer 0 clock source bit.
- 0 : The clock source of Timer 0 is SYSclk/12. It will compatible to the traditional 80C51 MCU
- 1 : The clock source of Timer 0 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

T1x12 : Timer 1 clock source bit.
- 0 : The clock source of Timer 1 is SYSclk/12. It will compatible to the traditional 80C51 MCU
- 1 : The clock source of Timer 1 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

UART_M0x6 : Baud rate select bit of UART1 while it is working under Mode-0
- 0 : The baud-rate of UART in mode 0 is SYSclk/12.
- 1 : The baud-rate of UART in mode 0 is SYSclk/2.

UART_M0x6 : Baud rate select bit of UART1 while it is working under Mode-0
- 0 : The baud-rate of UART in mode 0 is SYSclk/12.
- 1 : The baud-rate of UART in mode 0 is SYSclk/2.

EADCI : Enable/Disable interrupt from A/D converter
- 0 : (default) Inhibit the ADC functional block to generate interrupt to the MCU
- 1 : Enable the ADC functional block to generate interrupt to the MCU

ESPI : Enable/Disable interrupt from Serial Peripheral Interface (SPI)
- 0 : (default)Inhibit the SPI functional block to generate interrupt to the MCU
- 1 : Enable the SPI functional block to generate interrupt to the MCU

ELVDI : Enable/Disable interrupt from low-voltage sensor
- 0 : (default)Inhibit the low-voltage sensor functional block to generate interrupt to the MCU
- 1 : Enable the low-voltage sensor functional block to generate interrupt to the MCU

2. WAKE_CLKO: CLK_Output Power down Wake-up control register (Non bit-Addressable)

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

PCAWAKEUP: When set and the associated-PCA interrupt control registers is configured correctly, the CEXn pin of PCA function is enabled to wake up MCU from power-down state.

RXD_PIN_IE: When set and the associated-UART interrupt control registers is configured correctly, the RXD pin (P3.0) is enabled to wake up MCU from power-down state.

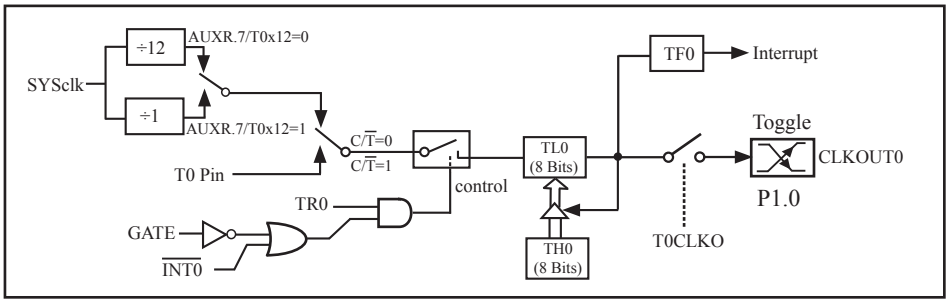
T1_PIN_IE : When set and the associated-Timer1 interrupt control registers is configured correctly, the T1 pin (P3.5) is enabled to wake up MCU from power-down state.

T0_PIN_IE : When set and the associated-Timer0 interrupt control registers is configured correctly, the T1 pin (P3.4) is enabled to wake up MCU from power-down state.

T1CKLO : When set, P1.1 is enabled to be the clock output of Timer 1. The clock rate is Timer 1overflow rate divided by 2.

T0CKLO : When set, P1.0 is enabled to be the clock output of Timer 0. The clock rate is Timer 0overflow rate divided by 2.

2.1.4.1 Timer 0 Programmable Clock-out on P1.0



Timer/Counter 0 Mode 2: 8-Bit Auto-Reload

STC12C2052AD is able to generate a programmable clock output on P1.0. When T0CLKO/WAKE_CLKO.0 bit in WAKE_CLKO SFR is set, T0 timer overflow pulse will toggle P1.0 latch to generate a 50% duty clock. The frequency of clock-out = T0 overflow rate/2.

If C/$\overline{T}$(TMOD.2) = 0, Timer/Counter 0 is set for Timer operation (input from internal system clock), the Frequency of clock-out is as following :

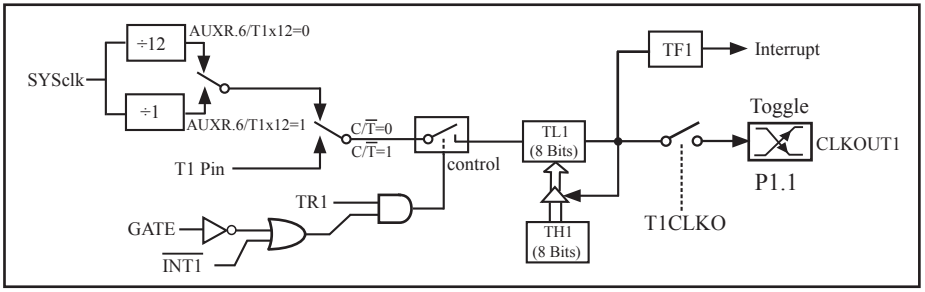
(SYSclk) / (256 – TH0) / 2, when AUXR.7 / T0x12=1

or **(SYSclk / 12) / (256 – TH0) / 2 , when AUXR.7 / T0x12=0**

If C/$\overline{T}$ (TMOD.2) = 1, Timer/Counter 0 is set for Conter operation (input from external P3.4/T0 pin), the Frequency of clock-out is as following :

T0_Pin_CLK / (256-TH0) / 2

2.1.4.2 Timer 1 Programmable Clock-out on P1.1



Timer/Counter 1 Mode 2: 8-Bit Auto-Reload

STC12C2052AD is able to generate a programmable clock output on P1.1. When T1CLKO/WAKE_CLKO.1 bit in WAKE_CLKO SFR is set, T1 timer overflow pulse will toggle P1.1 latch to generate a 50% duty clock. The frequency of clock-out = T1 overflow rate/2.

If C/$\overline{T}$(TMOD.6) = 0, Timer/Counter 1 is set for Timer operation (input from internal system clock), the Frequency of clock-out is as following :

(SYSclk) / (256 – TH1) / 2, when AUXR.6 / T0x12=1

or **(SYSclk / 12) / (256 – TH1) / 2 , when AUXR.6 / T0x12=0**

If C/$\overline{T}$(TMOD.6) = 1, Timer/Counter 1 is set for Conter operation (input from external P3.5/T1 pin), the Frequency of clock-out is as following :

T1_Pin_CLK / (256-TH1) / 2

2.2 Power Management Modes

The STC12C2052AD core has three software programmable power management mode: slow-down, idle and stop/power-down mode. The power consumption of STC12C2052AD series is about 2.7mA~7mA in normal operation, while it is lower than 0.1uA in stop/power-down mode and 1.8mA in idle mode.

Slow-down mode is controlled by clock divider register(CLK_DIV). Idle and stop/power-down is managed by the corresponding bit in Power control (PCON) register which is shown in below.

PCON register (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

SMOD : Double baud rate of UART interface

0 Keep normal baud rate when the UART is used in mode 1,2 or 3.

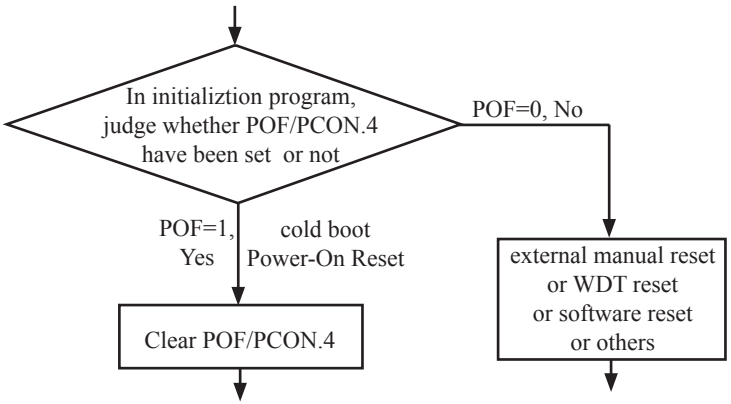
1 Double baud rate bit when the UART is used in mode 1,2 or 3.

SMOD0 : SM0/FE bit select for SCON.7; setting this bit will set SCON.7 as Frame Error function. Clearing it to set SCON.7 as one bit of UART mode selection bits.

LVDF : Pin Low-Voltage Flag. Once low voltage condition is detected (VCC power is lower than LVD voltage), it is set by hardware (and should be cleared by software).

POF : Power-On flag. It is set by power-off-on action and can only cleared by software.

Practical application: if it is wanted to know which reset the MCU is used, see the following figure.



GF1,GF0: General-purposed flag 1 and 0

PD : Stop Mode/Power-Down Select bit..

Setting this bit will place the STC12C2052AD MCU in Stop/Power-Down mode. Stop/Power-Down mode can be waked up by external interrupt. Because the MCU' s internal oscillator stopped in Stop/Power-Down mode, CPU, Timers, UARTs and so on stop to run, only external interrupt go on to work. The following pins can wake up MCU from Stop/Power-Down mode: $\overline{\text{INT0}}$/P3.2, $\overline{\text{INT1}}$/P3.3, T0/P3.4, T1/P3.5, RxD/P3.0, PCA0/PWM0/P3.7, PCA1/PWM1/P3.5, PCA2/PWM2/P2.0, PCA3/PWM3/P2.4.

IDL : Idle mode select bit.

Setting this bit will place the STC12C2052AD in Idle mode. only CPU goes into Idle mode. (Shuts off clock to CPU, but clock to Timers, Interrupts, Serial Ports, and Analog Peripherals are still active.) The following pins can wake up MCU from Idle mode: $\overline{\text{INT0}}$/P3.2, $\overline{\text{INT1}}$/P3.3, T0/P3.4, T1/P3.5, RxD/P3.0. Besides, Timer0 and Timer1 and UARTs interrupt also can wake up MCU from idle mode

2.2.1 Slow Down Mode

A divider is designed to slow down the clock source prior to route to all logic circuit. The operating frequency of internal logic circuit can therefore be slowed down dynamically , and then save the power.

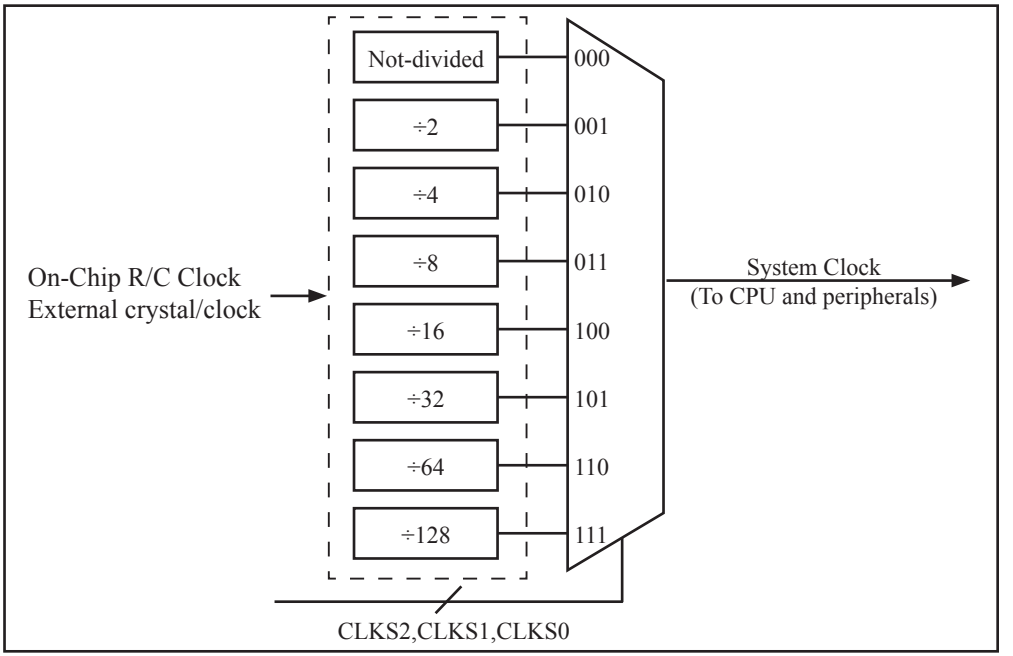
User can slow down the MCU by means of writing a non-zero value to the CLKS[2:0] bits in the CLK_DIV register. This feature is especially useful to save power consumption in idle mode as long as the user changes the CLKS[2:0] to a non-zero value before entering the idle mode.

CLK_DIV register (Clock Divider)

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

B2-B0 (CLKS2-CLKS0) :

000 External crystal/clock or On-Chip R/C clock is not divided (default state)
001 External crystal/clock or On-Chip R/C clock is divided by 2.
010 External crystal/clock or On-Chip R/C clock is divided by 4.
011 External crystal/clock or On-Chip R/C clock is divided by 8.
100 External crystal/clock or On-Chip R/C clock is divided by 16.
101 External crystal/clock or On-Chip R/C clock is divided by 32.
110 External crystal/clock or On-Chip R/C clock is divided by 64.
111 External crystal/clock or On-Chip R/C clock is divided by 128.



Clock Structure

2.2.2 Idle Mode

An instruction that sets IDL/PCON.0 causes that to be the last instruction executed before going into the idle mode, the internal clock is gated off to the CPU but not to the interrupt, timer, PCA, ADC, SPI, WDT and serial port functions. The PCA can be programmed either to pause or continue operating during Idle. The CPU status is preserved in its entirety: the RAM, Stack Pointer, Program Counter, Program Status Word, Accumulator, and all other registers maintain their data during Idle. The port pins hold the logical states they had at the time Idle was activated. Idle mode leaves the peripherals running in order to allow them to wake up the CPU when an interrupt is generated. Timer 0, Timer 1, PWM timer and UART will continue to function during Idle mode.

There are two ways to terminate the idle. Activation of any enabled interrupt will cause IDL/PCON.0 to be cleared by hardware, terminating the idle mode. The interrupt will be serviced, and following RETI, the next instruction to be executed will be the one following the instruction that put the device into idle.

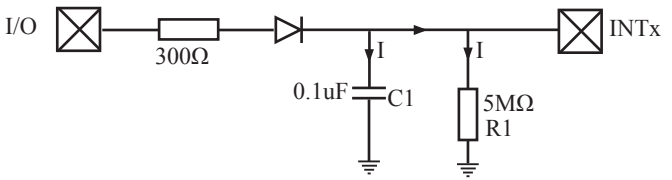
The flag bits (GF0 and GF1) can be used to give art indication if an interrupt occurred during normal operation or during Idle. For example, an instruction that activates Idle can also set one or both flag bits. When Idle is terminated by an interrupt, the interrupt service routine can examine the flag bits.

The other way to wake-up from idle is to pull RESET high to generate internal hardware reset.Since the clock oscillator is still running, the hardware reset neeeds to be held active for only two machine cycles(24 oscillator periods) to complete the reset.

2.2.3 Stop / Power Down (PD) Mode and Demo Program (C and ASM)

An instruction that sets PD/PCON.1 cause that to be the last instruction executed before going into the Power-Down mode. In the Power-Down mode, the on-chip oscillator and the Flash memory are stopped in order to minimize power consumption. Only the power-on circuitry will continue to draw power during Power-Down. The contents of on-chip RAM and SFRs are maintained. The power-down mode can be woken-up by RESET pin, external interrupt INT0 ~ INT1, RXD pin, T0 pin, T1 pin and PCA input pins—PWM pins and PWM pins. When it is woken-up by RESET, the program will execute from the address 0x0000. Be carefully to keep RESET pin active for at least 10ms in order for a stable clock. If it is woken-up from I/O, the CPU will rework through jumping to related interrupt service routine. Before the CPU rework, the clock is blocked and counted until 32768 in order for denouncing the unstable clock. To use I/O wake-up, interrupt-related registers have to be enabled and programmed accurately before power-down is entered. Pay attention to have at least one “NOP” instruction subsequent to the power-down instruction if I/O wake-up is used. When terminating Power-down by an interrupt, the wake up period is internally timed. At the negative edge on the interrupt pin, Power-Down is exited, the oscillator is restarted, and an internal timer begins counting. The internal clock will be allowed to propagate and the CPU will not resume execution until after the timer has reached internal counter full. After the timeout period, the interrupt service routine will begin. To prevent the interrupt from re-triggering, the interrupt service routine should disable the interrupt before returning. The interrupt pin should be held low until the device has timed out and begun executing. The user should not attempt to enter (or re-enter) the power-down mode for a minimum of 4 us until after one of the following conditions has occured: Start of code execution(after any type of reset), or Exit from power-down mode.

The following circuit can timing wake up MCU from power down mode when external interrupt sources do not exist



Operation step:

1. I/O ports are first configured to push-pull output(strong pull-up) mode
2. Writen 1s into ports I/O ports
3. the above circuit will charge the capacitor C1
4. Writen 0s into ports I/O ports, MCU will go into power-down mode
5. The above circuit will discharge. When the electricity of capacitor C1 has been discharged less than 0.8V, external interrupt INTx pin will generate a falling edge and wake up MCU from power-down mode automatically.

The following example C program demostrates that power-down mode be woken-up by external interrupt .

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ------------------------------------*/
/* --- STC 1T Series MCU wake up Power-Down mode Demo ------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/
#include <reg51.h>
#include <intrins.h>
sbit        Begin_LED = P1^2;                       //Begin-LED indicator indicates system start-up
unsigned char       Is_Power_Down = 0;              //Set this bit before go into Power-down mode
sbit        Is_Power_Down_LED_INT0          = P1^7; //Power-Down wake-up LED indicator on INT0
sbit        Not_Power_Down_LED_INT0         = P1^6; //Not  Power-Down wake-up LED indicator on INT0
sbit        Is_Power_Down_LED_INT1          = P1^5; //Power-Down wake-up LED indicator on INT1
sbit        Not_Power_Down_LED_INT1         = P1^4; //Not  Power-Down wake-up LED indicator on INT1
sbit        Power_Down_Wakeup_Pin_INT0      = P3^2; //Power-Down wake-up pin on INT0
sbit        Power_Down_Wakeup_Pin_INT1      = P3^3; //Power-Down wake-up pin on INT1
sbit        Normal_Work_Flashing_LED        = P1^3; //Normal work LED indicator
void  Normal_Work_Flashing (void);
void  INT_System_init (void);
void  INT0_Routine (void);
void  INT1_Routine (void);

void main (void)
{
        unsigned char       j = 0;
        unsigned char       wakeup_counter = 0;
                                        //clear interrupt wakeup counter variable wakeup_counter
        Begin_LED = 0;                  //system start-up LED
        INT_System_init ( );            //Interrupt system initialization
        while(1)
        {
                P2 = wakeup_counter;
                wakeup_counter++;
                for(j=0; j<2; j++)
                {
                        Normal_Work_Flashing( );    //System normal work
                }
```

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

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

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

```
void Normal_Work_Flashing (void)
{
        Normal_Work_Flashing_LED = 0;
        delay ( );
        Normal_Work_Flashing_LED = 1;
        delay ( );
}
```

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

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ----------------------------------*/
/* --- STC 1T Series MCU wake up Power-Down mode Demo ------*/
 /* If you want to use the program or the program referenced in the  */
 /* article, please specify in which data and procedures from STC    */
 /*---------------------------------------------------------------------------*/
;*************************************************************
;Wake Up Idle and Wake Up Power Down
;*************************************************************
                ORG     0000H
                AJMP    MAIN
                ORG     0003H
int0_interrupt:
                CLR     P1.7                    ;open P1.7 LED indicator
                ACALL   delay                   ;delay in order to observe
                CLR     EA                      ;clear global enable bit, stop all interrupts
                RETI

                ORG     0013H
int1_interrupt:
                CLR     P1.6                    ;open P1.6 LED indicator
                ACALL   delay                   ;;delay in order to observe
                CLR     EA                      ;clear global enable bit, stop all interrupts
                RETI
                ORG     0100H
delay:
                CLR     A
                MOV     R0,     A
                MOV     R1,     A
                MOV     R2,     #02
```

```
delay_loop:
                DJNZ    R0,     delay_loop
                DJNZ    R1,     delay_loop
                DJNZ    R2,     delay_loop
                RET
main:
                MOV     R3,     #0              ;P1 LED increment mode changed
                                                ;start to run program
main_loop:
                MOV     A,      R3
                CPL     A
                MOV     P1,     A
                ACALL   delay
                INC     R3
                MOV     A,      R3
                SUBB    A,      #18H
                JC      main_loop
                MOV     P1,     #0FFH           ;close all LED, MCU go into power-down mode
                CLR     IT0                     ;low electrical level trigger external interrupt 0
        ;       SETB    IT0                     ;negative edge trigger external interrupt 0
                SETB    EX0                     ;enable external interrupt 0
                CLR     IT1                     ;low electrical level trigger external interrupt 1
        ;       SETB    IT1                     ;negative edge trigger external interrupt 1
                SETB    EX1                     ;enable external interrupt 1
                SETB    EA                      ;set the global enable
                                                ;if don't so, power-down mode cannot be wake up

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

2.3 RESET Sources

In STC12C2052AD, there are 5 sources to generate internal reset. They are RST pin reset, software reset, On-chip power-on-reset(if delay 200mS after power-on reset, the reset mode is On-chip MAX810 POR timing delay which actully add 200mS delay after power-on reset), internal low-voltage detection reset and Watch-Dog-Timer reset.

2.3.1 RESET Pin

External RST pin reset accomplishes the MCU reset by forcing a reset pulse to RST pin from external. Asserting an active-high signal and keeping at least 24 cycles plus 10us on the RST pin generates a reset. If the signal on RST pin changed active-low level, MCU will end the reset state and start to run from the 0000H of user procedures.

2.3.2 Software RESET

Writing an "1" to SWRST bit in ISP_CONTR register will generate a internal reset.

ISP_CONTR: ISP/IAP Control Register

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

ISPEN : ISP/IAP operation enable.
0 : Global disable all ISP/IAP program/erase/read function.
1 : Enable ISP/IAP program/erase/read function.
SWBS: software boot selection control bit
0 : Boot from main-memory after reset.
1 : Boot from ISP memory after reset.
SWRST: software reset trigger control.
0 : No operation
1 : Generate software system reset. It will be cleared by hardware automatically.
CMD_FAIL: Command Fail indication for ISP/IAP operation.
0 : The last ISP/IAP command has finished successfully.
1 : The last ISP/IAP command fails. It could be caused since the access of flash memory was inhibited.

```
;Software reset from user appliction program area (AP area) and switch to AP area to run program
MOV ISP_CONTR, #00100000B ;SWBS = 0(Select AP area), SWRST = 1(Software reset)
;Software reset from system ISP monitor program area (ISP area) and switch to AP area to run program
MOV ISP_CONTR, #00100000B ;SWBS = 0(Select AP area), SWRST = 1(Software reset)
;Software reset from user appliction program area (AP area) and switch to ISP area to run program
MOV ISP_CONTR, #01100000B ;SWBS = 1(Select ISP area), SWRST = 1(Software reset)
;Software reset from system ISP monitor program area (ISP area) and switch to ISP area to run program
MOV ISP_CONTR, #01100000B ;SWBS = 1(Select ISP area), SWRST = 1(Software reset)
```

This reset is to reset the whole system, all special function registers and I/O prots will be reset to the initial value

2.3.3 Power-On Reset (POR)

When VCC drops below the detection threshold of POR circuit, all of the logic circuits are reset.

When VCC goes back up again, an internal reset is released automatically after a delay of 32768 clocks. The nominal POR detection threshold is around 1.9V for 3V device and 3.3V for 5V device.

The Power-On flag, POF/PCON.4, is set by hardware to denote the VCC power has ever been less than the POR voltage. And, it helps users to check if the start of running of the CPU is from power-on or from hardware reset (RST-pin reset), software reset or Watchdog Timer reset. The POF bit should be cleared by software.

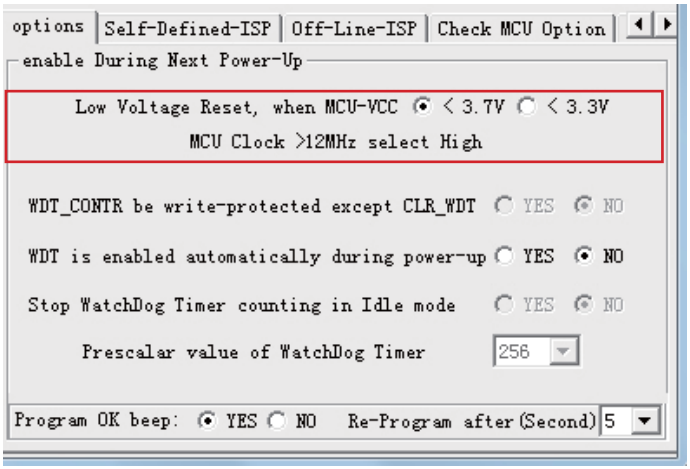
2.3.4 MAX810 power-on-Reset delay

There is another on-chip POR delay circuit s integrated on STC12C2052AD. This circuit is MAX810—sepcial reset circuit and is controlled by configuring STC-ISP Writter/Programmer shown in the next figure. MAX810 special reset circuit just add 200mS extra reset-delay-time after power-up reset. So it is another power-on reset.

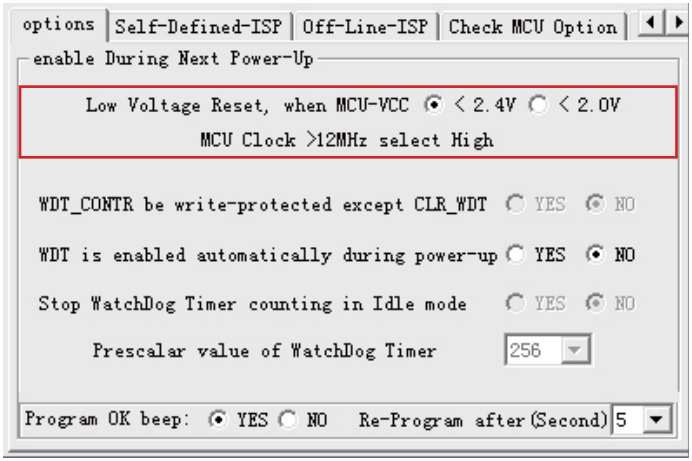
2.3.5 Internal Low Voltage Detection Reset

Besides the POR voltage, there is a higher threshold voltage: the Low Voltage Detection (LVD) voltage for STC12C2052AD series. When the VCC power drops down to the LVD voltage, the Low voltage Flag, LVDF bit (PCON.5), will be set by hardware. (Note that during power-up, this flag will also be set, and the user should clear it by software for the following Low Voltage detecting.) This flag can also generate an interrupt if the enable bit of LVD interrupt is set to 1.

The threshold voltage of STC12C2052AD built-in low voltage detection reset is optional in STC-ISP Writer/ Programmer. The detection voltage of 5V MCU of STC12C2052AD series is optional between 3.7V and 3.3V as shown in following figure.



The detection voltage of 3V MCU of STC12LE2052AD series is optional between 2.4V and 2.0V as shown in following figure.



Some SFRs related to Low voltage detection as shown below.

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

PCON register (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 : Pin Low-Voltage Flag. Once low voltage condition is detected (VCC power is lower than LVD voltage), it is set by hardware (and should be cleared by software).

AUXR: Auxiliary register (Non bit-addressable)

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

T0x12 : Timer 0 clock source bit.
0 : The clock source of Timer 0 is SYSclk/12. It will compatible to the traditional 80C51 MCU
1 : The clock source of Timer 0 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

T1x12 : Timer 1 clock source bit.
0 : The clock source of Timer 1 is SYSclk/12. It will compatible to the traditional 80C51 MCU
1 : The clock source of Timer 1 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

UART_M0x6 : Baud rate select bit of UART1 while it is working under Mode-0
0 : The baud-rate of UART in mode 0 is SYSclk/12.
1 : The baud-rate of UART in mode 0 is SYSclk/2.

EADCI : Enable/Disable interrupt from A/D converter
0 : (default) Inhibit the ADC functional block to generate interrupt to the MCU
1 : Enable the ADC functional block to generate interrupt to the MCU

ESPI : Enable/Disable interrupt from Serial Peripheral Interface (SPI)
0 : (default)Inhibit the SPI functional block to generate interrupt to the MCU
1 : Enable the SPI functional block to generate interrupt to the MCU

ELVDI : Enable/Disable interrupt from low-voltage sensor
0 : (default)Inhibit the low-voltage sensor functional block to generate interrupt to the MCU
1 : Enable the low-voltage sensor functional block to generate interrupt to the MCU

IE: Interrupt Enable Rsgister

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

Enable Bit = 1 enables the interrupt .
Enable Bit = 0 disables it .

EA (IE.7): disables all interrupts. if EA = 0,no interrupt will be acknowledged. if EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

EPCA_LVD (IE.6): Interrupt controller of Programmable Counter Array (PCA) and Low-Voltage Detector

0 : Disable theinterrupt of Programmable Counter Array (PCA) and Low-Voltage Detector

1 : enable theinterrupt of Programmable Counter Array (PCA) and Low-Voltage Detector

IPH: Interrupt Priority High Register

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

IP: Interrupt Priority Register

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

PPCA_LVDH, PPCA_LVD : Programmable Counter Array (PCA) and Low voltage detector interrupt priority control bits.

if PPCA_LVDH=0 and PPCA_LVD=0, Programmable Counter Array (PCA) and Low voltage detector interrupt are assigned lowest priority(priority 0).

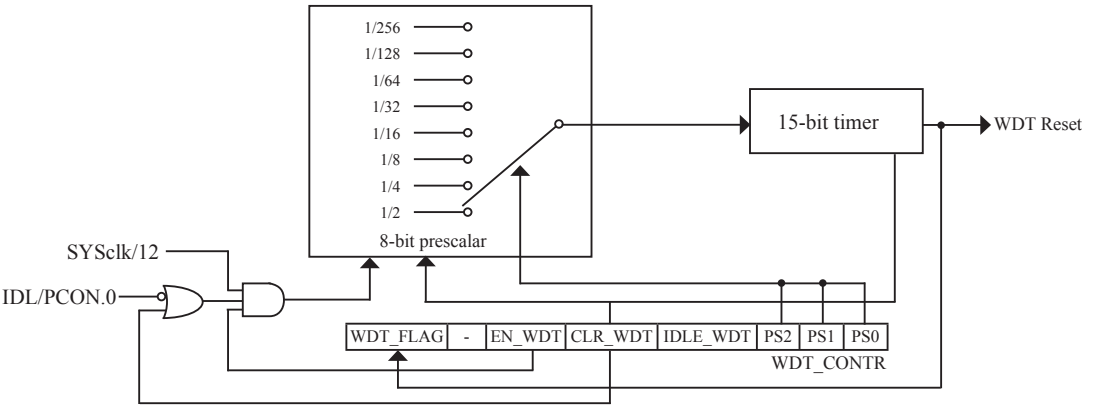
if PPCA_LVDH=0 and PPCA_LVD=1, Programmable Counter Array (PCA) and Low voltage detector interrupt are assigned lowest priority(priority 1).

if PPCA_LVDH=1 and PPCA_LVD=0, Programmable Counter Array (PCA) and Low voltage detector interrupt are assigned lowest priority(priority 2).

if PPCA_LVDH=1 and PPCA_LVD=1, Programmable Counter Array (PCA) and Low voltage detector interrupt are assigned lowest priority(priority 3).

2.3.6 Watch-Dog-Timer

The watch dog timer in STC12C2052AD consists of an 8-bit pre-scaler timer and an 15-bit timer. The timer is one-time enabled by setting EN_WDT(WDT_CONTR.5). Clearing EN_WDT can stop WDT counting. When the WDT is enabled, software should always reset the timer by writing 1 to CLR_WDT bit before the WDT overflows. If STC12C2052AD series MCU is out of control by any disturbance, that means the CPU can not run the software normally, then WDT may miss the "writting 1 to CLR_WDT" and overflow will come. An overflow of Watch-Dog-Timer will generate a internal reset.



WDT Structure

WDT_CONTR: Watch-Dog-Timer Control Register

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WDT_CONTR	0E1H	name	WDT_FLAG	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0

WDT_FLAG : WDT reset flag.
0 : This bit should be cleared by software.
1 : When WDT overflows, this bit is set by hardware to indicate a WDT reset happened.
EN_WDT : Enable WDT bit. When set, WDT is started.
CLR_WDT : WDT clear bit. When set, WDT will recount. Hardware will automatically clear this bit.
IDLE_WDT : WDT IDLE mode bit. When set, WDT is enabled in IDLE mode. When clear, WDT is disabled in IDLE.

PS2, PS1, PS0 : WDT Pre-scale value set bit.
Pre-scale value of Watchdog timer is shown as the bellowed table :

PS2	PS1	PS0	Pre-scale	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 overflow time is determined by the following equation:

WDT overflow time = (12 × Pre-scale × 32768) / SYSclk

The SYSclk is 20MHz in the table above.
If SYSclk is 12MHz, The WDT overflow time is :

WDT overflow time = (12 × Pre-scale × 32768) / 12000000 = Pre-scale× 393216 / 12000000

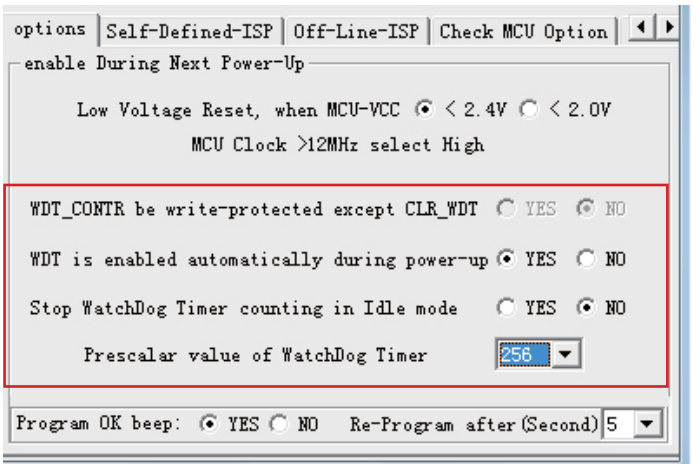
WDT overflow time is shown as the bellowed table when SYSclk is 12MHz:

PS2	PS1	PS0	Pre-scale	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

WDT overflow time is shown as the bellowed table when SYSclk is 11.0592MHz:

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

Options related with WDT in STC-ISP Writter/Programmer is shown in the following figure



The following example is a assembly language program that demostrates STC 1T Series MCU WDT.

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------*/
;/* --- STC 1T Series MCU WDT Demo ---------------------------------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/
; WDT overflow time = (12 × Pre-scale × 32768) / SYSclk
WDT_CONTR               EQU     0E1H            ;WDT address
WDT_TIME_LED            EQU     P1.5            ;WDT overflow time LED on P1.5
                                                ;The WDT overflow time may be measured by the LED light time
WDT_FLAG_LED            EQU     P1.7
                                                ;WDT overflow reset flag LED indicator on P1.7
Last_WDT_Time_LED_Status        EQU     00H
                        ;bit variable used to save the last stauts of WDT overflow time LED indicator

;WDT reset time , the SYSclk is 18.432MHz
;Pre_scale_Word  EQU     00111100B       ;open WDT, Pre-scale value is 32, WDT overflow time=0.68S
;Pre_scale_Word  EQU     00111101B       ;open WDT, Pre-scale value is 64, WDT overflow time=1.36S
;Pre_scale_Word  EQU     00111110B       ;open WDT, Pre-scale value is 128, WDT overflow time=2.72S
;Pre_scale_Word  EQU     00111111B       ;open WDT, Pre-scale value is 256, WDT overflow time=5.44S

                ORG     0000H
                AJMP    MAIN
                ORG     0100H
MAIN:
                MOV     A,      WDT_CONTR               ;detection if WDT reset
                ANL     A,      #10000000B
                JNZ     WDT_Reset
                        ;WDT_CONTR.7=1, WDT reset, jump WDT reset subroutine
                        ;WDT_CONTR.7=0, Power-On reset, cold start-up, the content of RAM is  random
                SETB    Last_WDT_Time_LED_Status        ;Power-On reset
                CLR     WDT_TIME_LED            ;Power-On reset,open WDT overflow time LED
                MOV     WDT_CONTR,      #Pre_scale_Word         ;open WDT
```

```
WAIT1:
        SJMP    WAIT1                   ;wait WDT overflow reset
    ;WDT_CONTR.7=1, WDT reset, hot strart-up, the content of RAM is constant and just like before reset
WDT_Reset:
        CLR     WDT_FLAG_LED
                                        ;WDT reset,open WDT overflow reset flag LED indicator
        JB      Last_WDT_Time_LED_Status,       Power_Off_WDT_TIME_LED
        ;when set Last_WDT_Time_LED_Status, close the  corresponding LED indicator
        ;clear, open the  corresponding LED indicator
        ;set WDT_TIME_LED according to the last status of WDT overflow time LED indicator
        CLR     WDT_TIME_LED            ;close the WDT overflow time LED indicator
        CPL     Last_WDT_Time_LED_Statu
                                        ;reverse the last status of WDT overflow time LED indicator

WAIT2:
        SJMP    WAIT2                           ;wait WDT overflow reset
Power_Off_WDT_TIME_LED:
        SETB    WDT_TIME_LED            ;close the WDT overflow time LED indicator
        CPL     Last_WDT_Time_LED_Status
                                        ;reverse the last status of WDT overflow time LED indicator
WAIT3:
        SJMP    WAIT3                           ;wait WDT overflow reset
        END
```

2.3.7 Warm Boot and Cold Boot Reset

Reset type	Reset source	Result
Warm boot	WatchDog	System will reset to AP address 0000H and begin running user application program
	Reset Pin	
	20H → ISP_CONTR	
	60H → ISP_CONTR	System will reset to ISP address 0000H and begin running ISP monitor program, if not detected legitimate ISP command, system will software reset to the user program area automatically.
Cold boot	Power-on	

Chapter 3. Memory Organization

The STC12C2052AD series MCU has separate address space for Program Memory and Data Memory. The logical separation of program and data memory allows the data memory to be accessed by 8-bit addresses, which can be quickly stored and manipulated by the CPU.

Program memory (ROM) can only be read, not written to. In the STC12C2052AD series, all the program memory are on-chip Flash memory, and without the capability of accessing external program memory because of no External Access Enable (/EA) and Program Store Enable (/PSEN) signals designed.

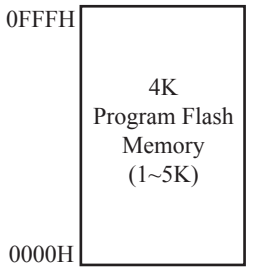
Data memory occupies a separate address space from program memory. In the 12C5A60S2 series, there are 256 bytes internal scratch-pad RAM, the high 128 bytes of which seemingly overlap with SFRs in address. Actually, they are distinguished by different addressing way. Similarly, STC12C2052AD series also have no the capability of accessing external data memory because of no the bus that access external data memory.

3.1 Program Memory

Program memory is the memory which stores the program codes for the CPU to execute. There is 1K/2K/3K/4K/5K-bytes of flash memory embedded for program and data storage. The design allows users to configure it as like there are three individual partition banks inside. They are called AP(application program) region, IAP (In-Application-Program) region and ISP (In-System-Program) boot region. AP region is the space that user program is resided. IAP(In-Application-Program) region is the nonvolatile data storage space that may be used to save important parameters by AP program. In other words, the IAP capability of STC12C2052AD provides the user to read/write the user-defined on-chip data flash region to save the needing in use of external EEPROM device. ISP boot region is the space that allows a specific program we calls “ISP program” is resided. Inside the ISP region, the user can also enable read/write access to a small memory space to store parameters for specific purposes. Generally, the purpose of ISP program is to fulfill AP program upgrade without the need to remove the device from system. STC12C2052AD hardware catches the configuration information since power-up duration and performs out-of-space hardware-protection depending on pre-determined criteria. The criteria is AP region can be accessed by ISP program only, IAP region can be accessed by ISP program and AP program, and ISP region is prohibited access from AP program and ISP program itself. But if the “ISP data flash is enabled”, ISP program can read/write this space. When wrong settings on ISP-IAP SFRs are done, The “out-of-space” happens and STC12C2052AD follows the criteria above, ignore the trigger command.

After reset, the CPU begins execution from the location 0000H of Program Memory, where should be the starting of the user’s application code. To service the interrupts, the interrupt service locations (called interrupt vectors) should be located in the program memory. Each interrupt is assigned a fixed location in the program memory. The interrupt causes the CPU to jump to that location, where it commences execution of the service routine. External Interrupt 0, for example, is assigned to location 0003H. If External Interrupt 0 is going to be used, its service routine must begin at location 0003H. If the interrupt is not going to be used, its service location is available as general purpose program memory.

The interrupt service locations are spaced at an interval of 8 bytes: 0003H for External Interrupt 0, 000BH for Timer 0, 0013H for External Interrupt 1, 001BH for Timer 1, etc. If an interrupt service routine is short enough (as is often the case in control applications), it can reside entirely within that 8-byte interval. Longer service routines can use a jump instruction to skip over subsequent interrupt locations, if other interrupts are in use.



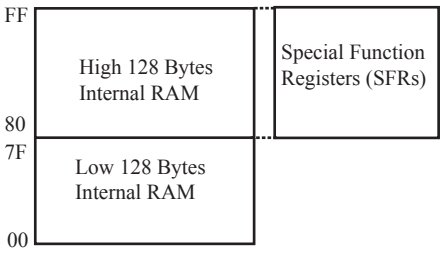
STC12C4052AD MCU Program Memory

Type	Program Memory
STC12C/LE1052AD	0000H~03FFH (1K)
STC12C/LE2052AD	0000H~07FFH (2K)
STC12C/LE3052AD	0000H~0BFFH (3K)
STC12C/LE4052AD	0000H~0FFFH (4K)
STC12CLE5052AD	0000H~13FFH (5K)

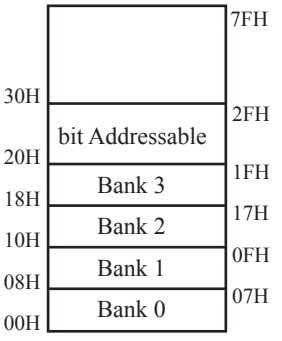
3.2 Data Memory(SRAM)

Just the same as the conventional 8051 micro-controller, there are 256 bytes of SRAM data memory plus 128 bytes of SFR space available on the STC12C2052AD. The lower 128 bytes of data memory may be accessed through both direct and indirect addressing. The upper 128 bytes of data memory and the 128 bytes of SFR space share the same address space. The upper 128 bytes of data memory may only be accessed using indirect addressing. The 128 bytes of SFR can only be accessed through direct addressing. The lowest 32 bytes of data memory are grouped into 4 banks of 8 registers each. Program instructions call out these registers as R0 through R7. The RS0 and RS1 bits in PSW register select which register bank is in use. Instructions using register addressing will only access the currently specified bank. This allows more efficient use of code space, since register instructions are shorter than instructions that use direct addressing. The next 16 bytes (20H~2FH) above the register banks form a block of bit-addressable memory space. The 80C51 instruction set includes a wide selection of single-bit instructions, and the 128 bits in this area can be directly addressed by these instructions. The bit addresses in this area are 00H through 7FH.

All of the bytes in the Lower 128 can be accessed by either direct or indirect addressing while the Upper 128 can only be accessed by indirect addressing. SFRs include the Port latches, timers, peripheral controls, etc. These registers can only be accessed by direct addressing. Sixteen addresses in SFR space are both byte- and bit-addressable. The bit-addressable SFRs are those whose address ends in 0H or 8H.



On-chip Scratch-Pad RAM



Lower 128 Bytes of internal SRAM

PSW register

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

CY : Carry flag.
This bit is set when the last arithmetic operation resulted in a carry (addition) or a borrow (subtrac-tion). It is cleared to logic 0 by all other arithmetic operations.

AC : Auxilliary Carry Flag.(For BCD operations)
This bit is set when the last arithmetic operation resulted in a carry into (addition) or a borrow from (subtraction) the high order nibble. It is cleared to logic 0 by all other arithmetic operations

F0 : Flag 0.(Available to the user for general purposes)

RS1: Register bank select control bit 1.
RS0: Register bank select control bit 0.
[RS1 RS0] select which register bank is used during register accesses

RS1	RS0	Working Register Bank(R0~R7) and Address
0	0	Bank 0(00H~07H)
0	1	Bank 1(08H~0FH)
1	0	Bank 2(10H~17H)
1	1	Bank 3(18H~1FH)

OV : Overflow flag.
This bit is set to 1 under the following circumstances:
• An ADD, ADDC, or SUBB instruction causes a sign-change overflow.
• A MUL instruction results in an overflow (result is greater than 255).
• A DIV instruction causes a divide-by-zero condition.
The OV bit is cleared to 0 by the ADD, ADDC, SUBB, MUL, and DIV instructions in all other cases.

F1 : Flag 1. User-defined flag.

P : Parity flag.
This bit is set to logic 1 if the sum of the eight bits in the accumulator is odd and cleared if the sum is even.

SP : Stack Pointer.

The Stsek Pointer Register is 8 bits wide. It is incremented before data is stored during PUSH and CALL executions. The stack may reside anywhere in on-chip RAM.On reset, the Stack Pointer is initialized to 07H causing the stack to begin at location 08H, which is also the first register (R0) of register bank 1. Thus, if more than one register bank is to be used, the SP should be initialized to a location in the data memory not being used for data storage. The stack depth can extend up to 256 bytes.

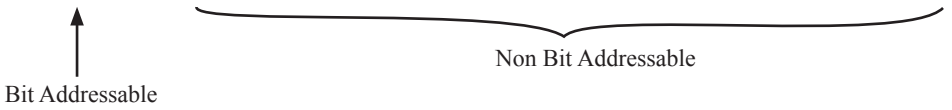
;Demo Program of internal common 256 bytes RAM

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------*/
;/* --- STC 1T Series MCU internal common 256 bytes RAM Demo ---*/
;/* If you want to use the program or the program referenced in the  ----*/
;/* article, please specify in which data and procedures from STC    ----*/
;/*---------------------------------------------------------------------------*/
TEST_CONST      EQU     5AH
;TEST_RAM       EQU     03H
        ORG     0000H
        LJMP    INITIAL

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

        END
```

3.3 Special Function Registers

3.3.1 Special Function Registers Address Map

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



3.3.2 Special Function Registers Bits Description

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
P0	Port 0	80H	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0	1111 1111B
SP	Stack Pointer	81H									0000 0111B
DPTR DPL	Data Pointer Low	82H									0000 0000B
DPTR DPH	Data Pointer High	83H									0000 0000B
SPSTAT	SPI Status register	84H	SPIF	WCOL	-	-	-	-	-	-	00xx xxxxB
SPCTL	SPI control register	85H	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	0000 0100B
SPDAT	SPI Data register	86H									0000 0000B
PCON	Power Control	87H	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011 0000B
TCON	Timer Control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
TMOD	Timer Mode	89H	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	0000 0000B
TL0	Timer Low 0	8AH									0000 0000B
TL1	Timer Low 1	8BH									0000 0000B
TH0	Timer High 0	8CH									0000 0000B
TH1	Timer High 1	8DH									0000 0000B
AUXR	Auxiliary register	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
WAKE_CLKO	CLK_Output Power down Wake-up control register	8FH	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO	0000 xx00B
P1	Port 1	90H	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	1111 1111B
P1M0	P1 configuration 0	91H									0000 0000B
P1M1	P1 configuration 1	92H									0000 0000B
SCON	Serial Control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000 0000B
SBUF	Serial Buffer	99H									xxxx xxxxB
P2	Port 2	A0H	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	1111 1111B
IE	Interrupt Enable	A8H	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0	0000 0000B
SADDR	Slave Address	A9H									0000 0000B
P3	Port 3	B0H	P3.7	-	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0	1x11 1111B
P3M0	P2 configuration 0	B1H									0000 0000B
P3M1	P3 configuration 1	B2H									0000 0000B
IPH	Interrupt Priority High	B7H	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H	x000 0000B
IP	Interrupt Priority Low	B8H	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0	x000 0000B
SADEN	Slave Address Mask	B9H									0000 0000B
ADC_CONTR	ADC Control	C5H	ADC_POWER	SPEED1	SPEED0	ADC_FLAG	ADC_START	CHS2	CHS1	CHIS0	0000 0000B
ADC_DATA	ADC Result High	C6H									0000 0000B

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
CLK_DIV	Clock Divder	C7h	-	-	-	-	-	CLKS2	CLKS1	CLKS0	xxxx x000B
PSW	Program Status Word	D0H	CY	AC	F0	RS1	RS0	OV	F1	P	0000 0000B
CCON	PCA Control Register	D8H	CF	CR	-	-	-	-	CCF1	CCF0	00xx xx00B
CMOD	PCA Mode Register	D9H	CIDL	-	-	-	-	CPS1	CPS0	ECF	00xx x000B
CCAPM0	PCA Module 0 Mode Register	DAH	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0	x000 0000B
CCAPM1	PCA Module 1 Mode Register	DBH	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1	x000 0000B
ACC	Accumulator	E0H									0000 0000B
WDT_CONTR	Watch-Dog-Timer Control Register	E1H	WDT_FLAG	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0	xx00 0000B
ISP_DATA	ISP/IAP Flash Data Register	E2H									1111 1111B
ISP_ADDRH	ISP/IAP Flash Address High	E3H									0000 0000B
ISP_ADDRL	ISP/IAP Flash Address Low	E4H									0000 0000B
ISP_CMD	ISP/IAP Flash Command Register	E5H	-	-	-	-	-	-	MS1	MS0	xxxx x000B
ISP_TRIG	ISP/IAP Flash Command Trigger	E6H									xxxx xxxxB
ISP_CONTR	ISP/IAP Control Register	E7H	IAPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0	0000 x000B
CL	PCA Base Timer Low	E9H									0000 0000B
CCAP0L	PCA module 0 capture register low	EAH									0000 0000B
CCAP1L	PCA module 1 capture register low	EBH									0000 0000B
B	B Register	F0H									0000 0000B
PCA_PWM0	PCA PWM mode auxiliary register 1	F2H	-	-	-	-	-	-	EPC0H	EPC0L	xxxx xx00B
PCA_PWM1	PCA PWM mode auxiliary register 1	F3H	-	-	-	-	-	-	EPC1H	EPC1L	xxxx xx00B
CH	PCA Base Timer High	F9H									0000 0000B
CCAP0H	PCA module 0 capture register high	FAH									0000 0000B
CCAP1H	PCA module 1 capture register high	FBH									0000 0000B

Some common SFRs of standard 8051 are shown as below.

Accumulator

ACC is the Accumulator register. The mnemonics for accumulator-specific instructions, however, refer to the accumulator simply as A.

B-Register

The B register is used during multiply and divide operations. For other instructions it can be treated as another scratch pad register.

Stack Pointer

The Stack Pointer register is 8 bits wide. It is incrementde before data is stored during PUSH and CALL executions. While the stack may reside anywhee in on-chip RAM, the Stack Pointer is initialized to 07H after a reset. Therefore, the first value pushed on the stack is placed at location 0x08, which is also the first register (R0) of register bank 1. Thus, if more than one register bank is to be used, the SP should be initialized to a location in the data memory not being used for data storage. The stack depth can extend up to 256 bytes.

Data Pointer Register (DPTR)

The Data Pointer (DPTR) consists of a high byte (DPH) and a low byte (DPL). Its intended function is to hold a 16-bit address. It may be manipulated as a 16-bit register or as two independent 8-bit registers.

Program Status Word(PSW)

The program status word(PSW) contains several status bits that reflect the current state of the CPU. The PSW, shown below, resides in the SFR space. It contains the Carry bit, the Auxiliary Carry(for BCD operation), the two register bank select bits, the Overflow flag, a Parity bit and two user-definable status flags.

The Carry bit, other than serving the function of a Carry bit in arithmetic operations, also serves as the “Accumulator” for a number of Boolean operations.

The bits RS0 and RS1 are used to select one of the four register banks shown in the previous page. A number of instructions refer to these RAM locations as R0 through R7.

The Parity bit reflects the number of 1s in the Accumulator. P=1 if the Accumulator contains an odd number of 1s and otherwise P=0.

PSW register

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

CY : Carry flag.
This bit is set when the last arithmetic operation resulted in a carry (addition) or a borrow (subtrac-tion). It is cleared to logic 0 by all other arithmetic operations.

AC : Auxilliary Carry Flag.(For BCD operations)
This bit is set when the last arithmetic operation resulted in a carry into (addition) or a borrow from (subtraction) the high order nibble. It is cleared to logic 0 by all other arithmetic operations

F0 : Flag 0.(Available to the user for general purposes)

RS1: Register bank select control bit 1.
RS0: Register bank select control bit 0.
[RS1 RS0] select which register bank is used during register accesses

RS1	RS0	Working Register Bank(R0~R7) and Address
0	0	Bank 0(00H~07H)
0	1	Bank 1(08H~0FH)
1	0	Bank 2(10H~17H)
1	1	Bank 3(18H~1FH)

OV : Overflow flag.
This bit is set to 1 under the following circumstances:
• An ADD, ADDC, or SUBB instruction causes a sign-change overflow.
• A MUL instruction results in an overflow (result is greater than 255).
• A DIV instruction causes a divide-by-zero condition.
The OV bit is cleared to 0 by the ADD, ADDC, SUBB, MUL, and DIV instructions in all other cases.

F1 : Flag 1. User-defined flag.

P : Parity flag.
This bit is set to logic 1 if the sum of the eight bits in the accumulator is odd and cleared if the sum is even.

Chapter 4. Configurable I/O Ports of STC12C2052AD series

4.1 I/O Ports Configurations

All I/O ports of STC12C2052AD may be independently configured to one of four modes by setting the corresponding bit in two mode registers PxMn (x= 0 ~ 3, n = 0, 1).The four modes are quasi-bidirectional (standard 8051 port output), push-pull output, input-only or open-drain output. All port pins default to quasi-bidirectional after reset. Each one has a Schmitt-triggered input for improved input noise rejection. Any port can drive 20mA current, but the whole chip had better drive lower than 90mA current.

Configure I/O ports mode

P3 Configure <P3.7, P3.6, P3.5, P3.4, P3.3, P3.2, P3.1, P3.0 port> (P3 address：B0H)

P3M0[7 : 0]	P3M1 [7 : 0]	I/O ports Mode
0	0	quasi_bidirectional(standard 8051 I/O port output） , Sink Current up to 20mA , pull-up Current is 230μA , Because of manufactured error, the actual pull-up current is 250uA ~ 150uA
0	1	push-pull output(strong pull-up output， current can be up to 20mA, resistors need to be added to restrict current
1	0	input-only (high-impedance)
1	1	Open Drain， internal pull-up resistors should be disabled and external pull-up resistors need to join.

```
Example:  MOV   P3M0, #10100000B
          MOV   P3M1, #10010000B
```

;P3.7 in Open Drain mode, P3.5 in high-impedance input, P3.4 in strong push-pull output, P3.3/P3.2/P3.1/P3.0 in quasi_bidirectional/weak pull-up

P1 Configure <P1.7, P1.6, P1.5, P1.4, P1.3, P1.2, P1.1, P1.0 port> (P1 address：90H)

P1M0[7 : 0]	P1M1 [7 : 0]	I/O ports Mode
0	0	quasi_bidirectional(standard 8051 I/O port output） , Sink Current up to 20mA , pull-up Current is 230μA , Because of manufactured error, the actual pull-up current is 250uA ~ 150uA
0	1	push-pull output(strong pull-up output， current can be up to 20mA, resistors need to be added to restrict current
1	0	input-only (high-impedance)
1	1	Open Drain， internal pull-up resistors should be disabled and external pull-up resistors need to join.

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

;P1.7 in Open Drain mode, P1.6 in strong push-pull output, P1.5 in high-impedance input, P1.4/P1.3/P1.2/ P1.1/P1.0 in quasi_bidirectional/weak pull-up

Some SFRs related with I/O ports are listed below.

P3 register (bit addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3	B0H	name	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0

P3 register could be bit-addressable and set/cleared by CPU. And P3.7~P3.0 coulde be set/cleared by CPU.

P3M0 register (non bit addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3M0	B1H	name	P3M0.7	P3M0.6	P3M0.5	P3M0.4	P3M0.3	P3M0.2	P3M0.1	P3M0.0

P3M1 register (non bit addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P3M1	B2H	name	P3M1.7	P3M1.6	P3M1.5	P3M1.4	P3M1.3	P3M1.2	P3M1.1	P3M1.0

P1 register (bit addressable)

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

P1 register could be bit-addressable and set/cleared by CPU. And P1.7~P1.0 coulde be set/cleared by CPU.

P1M0 register (non bit addressable)

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

P1M1 register (non bit addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
P1M1	92H	name	P1M1.7	P1M1.6	P1M1.5	P1M0.4	P1M1.3	P1M1.2	P1M1.1	P1M1.0

4.2 I/O ports Modes

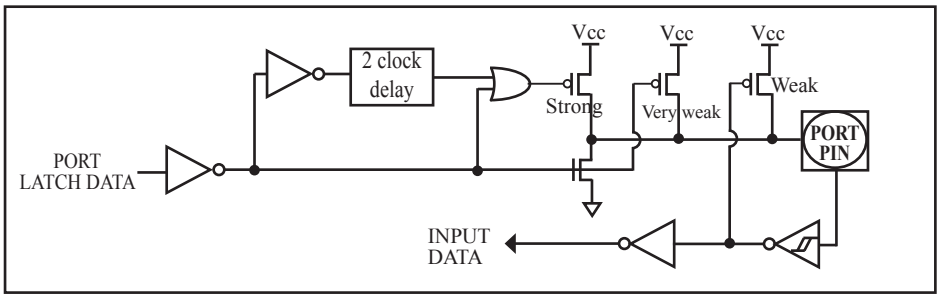
4.2.1 Quasi-bidirectional I/O

Port pins in quasi-bidirectional output mode function similar to the standard 8051 port pins. A quasi-bidirectional port can be used as an input and output without the need to reconfigure the port. This is possible because when the port outputs a logic high, it is weakly driven, allowing an external device to pull the pin low. When the pin outputs low, it is driven strongly and able to sink a large current. There are three pull-up transistors in the quasi-bidirectional output that serve different purposes.

One of these pull-ups, called the “very weak” pull-up, is turned on whenever the port register for the pin contains a logic “1”. This very weak pull-up sources a very small current that will pull the pin high if it is left floating.

A second pull-up, called the “weak” pull-up, is turned on when the port register for the pin contains a logic “1” and the pin itself is also at a logic “1” level. This pull-up provides the primary source current for a quasi-bidirectional pin that is outputting a 1. If this pin is pulled low by the external device, this weak pull-up turns off, and only the very weak pull-up remains on. In order to pull the pin low under these conditions, the external device has to sink enough current to over-power the weak pull-up and pull the port pin below its input threshold voltage.

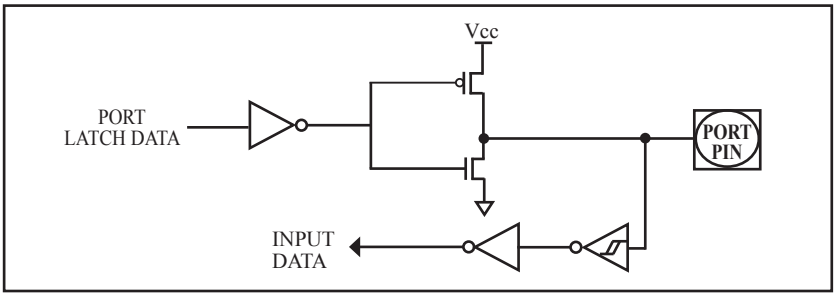
The third pull-up is referred to as the “strong” pull-up. This pull-up is used to speed up low-to-high transitions on a quasi-bidirectional port pin when the port register changes from a logic “0” to a logic “1”. When this occurs, the strong pull-up turns on for two CPU clocks, quickly pulling the port pin high.



Quasi-bidirectional output

4.2.2 Push-pull Output

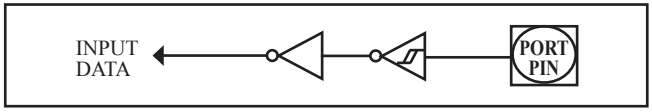
The push-pull output configuration has the same pull-down structure as both the open-drain and the quasi-bidirectional output modes, but provides a continuous strong pull-up when the port register conatins a logic “1”. The push-pull mode may be used when more source current is needed from a port output. In addition, input path of the port pin in this configuration is also the same as quasi-bidirectional mode.



Push-pull output

4.2.3 Input-only (High-Impedance)Mode

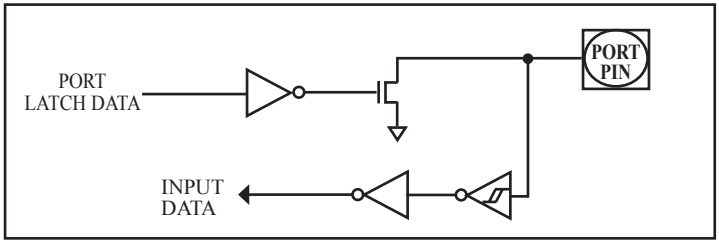
The input-only configuration is a Schmitt-triggered input without any pull-up resistors on the pin.



Input-only Mode

4.2.4 Open-drain Output

The open-drain output configuration turns off all pull-ups and only drives the pull-down transistor of the port pin when the port register contains a logic “0”. To use this configuration in application, a port pin must have an external pull-up, typically tied to VCC. The input path of the port pin in this configuration is the same as quasi-bidirection mode.



Open-drain output

4.3 I/O port application notes

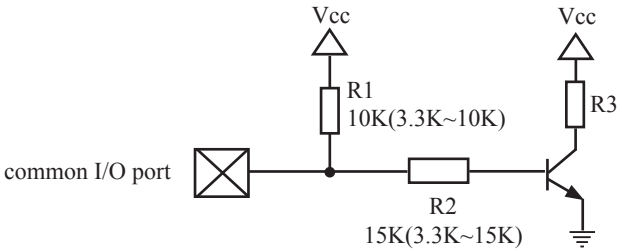
Traditional 8051 access I/O (signal transition or read status) timing is 12 clocks, STC12C2052AD series MCU is 4 clocks. When you need to read an external signal, if internal output a rising edge signal, for the traditional 8051, this process is 12 clocks, you can read at once, but for STC12C2052AD series MCU, this process is 4 clocks, when internal instructions is complete but external signal is not ready, so you must delay 1~2 nop operation.

When MCU is connected to a SPI or I2C or other open-drain peripherals circuit, you need add a 10K pull-up resistor.

Some IO port connected to a PNP transistor, but no pul-up resistor. The correct access method is IO port pull-up resistor and transistor base resistor should be consistent, or IO port is set to a strongly push-pull output mode.

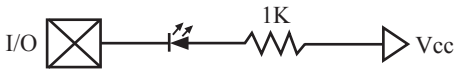
Using IO port drive LED directly or matrix key scan, needs add a 470ohm to 1Kohm resistor to limit current.

4.4 Typical transistor control circuit



If I/O is configed as “weak” pull-up, you should add a external pull-up resistor R1(3.3K~10K ohm). If no pull-up resistor R1, proposal to add a 15K ohm series resistor R2 at least or config I/O as “push-pull” mode.

4.5 Typical diode control circuit

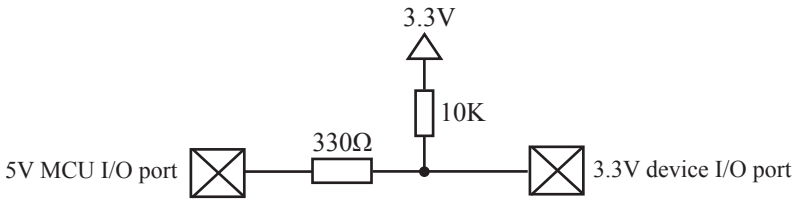


For weak pull-up / quasi-bidirectional I/O, use sink current drive LED, current limiting resistor as greater than 1K ohm, minimum not less than 470 ohm.

For push-pull / strong pull-up I/O, use drive current drive LED.

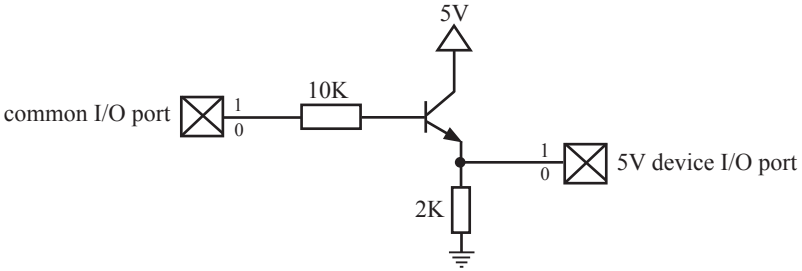
4.6 3V/5V hybrid system

When STC12C2052AD series 5V MCU connect to 3.3V peripherals. To prevent the 3.3V device can not afford to 5V voltage, the 5V MCU corresponding I/O should first add a 330 ohm current limiting resistor to 3.3 device I/O ports. And in intialization of procedures the 5V MCU corresponding I/O is set to open drain mode, disconnect the internal pull-up resistor, the corresponding 3.3V device I/O port add 10K ohm external pull-up resistor to the 3.3V device VCC, so high level to 3.3V and low to 0V, which can proper functioning



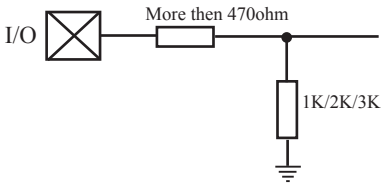
When STC12LE2052AD series 3V MCU connect to 5V peripherals. To prevent the 3V MCU can not afford to 5V voltage, if the corresponding I/O port as input port, the port may be in an isolation diode in series, isolated high-voltage part. When the external signal is higher than MCU operating voltage, the diode cut-off, I/O have been pulled high by the internal pull-up resistor; when the external signal is low, the diode conduction, I/O port voltage is limited to 0.7V, it's low signal to MCU.

When STC12LE2052AD series 3V MCU connect to 5V peripherals. To prevent the 3V MCU can not afford to 5V voltage, if the corresponding I/O port as output port, the port may be connect a NPN transistor to isolate high-voltage part. The circuit is shown as below.



4.7 How to make I/O port low after MCU reset

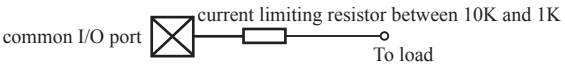
Traditional 8051 MCU power-on reset, the general IO port are weak pull-high output, while many practical applications require IO port remain low level after power-on reset, otherwise the system malfunction would be generated. For STC12C2052AD series MCU, IO port can add a pull-down resistor (1K/2K/3K), so that when power-on reset, although a weak internal pull-up to make MCU output high, but because of the limited capacity of the internal pull-up, it can not pull-high the pad, so this IO port is low level after power-on reset. If the I/O port need to drive high, you can set the IO model as the push-pull output mode, while the push-pull mode the drive current can be up to 20mA, so it can drive this I/O high.



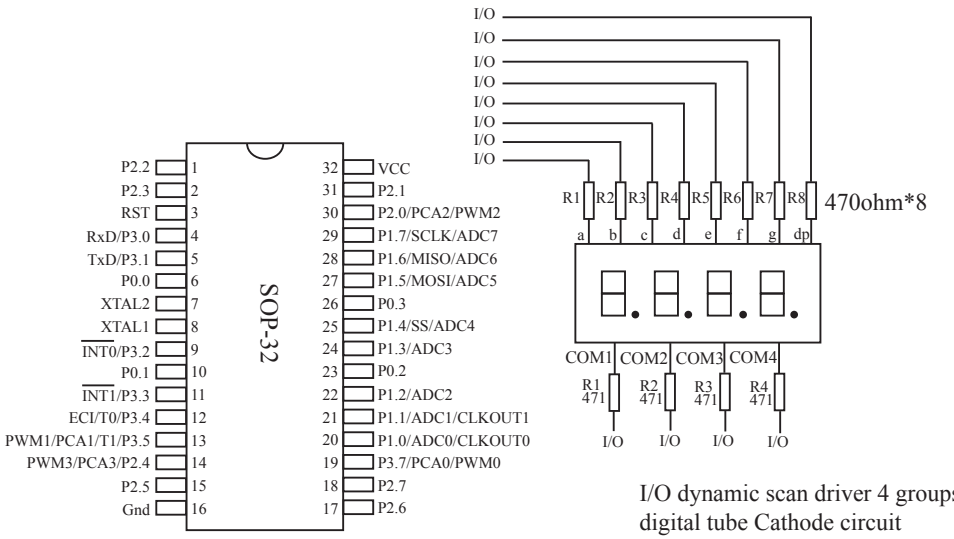
4.8 I/O status while PWM outputing

When I/O is used as PWM port, it's status as bellow:

Before PWM output	While PWM outputing
Quasi-bidirectional	Push-Pull (Strong pull-high need 1K~10K limiting resistor)
Push-Pull	Push-Pull (Strong pull-high need 1K~10K limiting resistor)
Input ony (Floating)	PWM Invalid
Open-drain	Open-drain

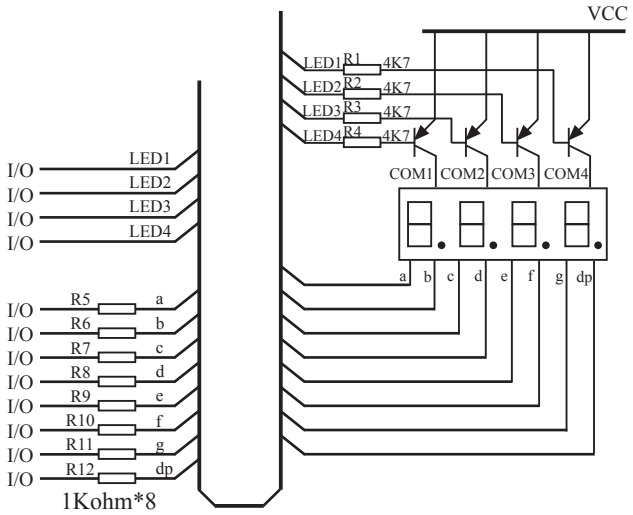


4.9 I/O drive LED application circuit

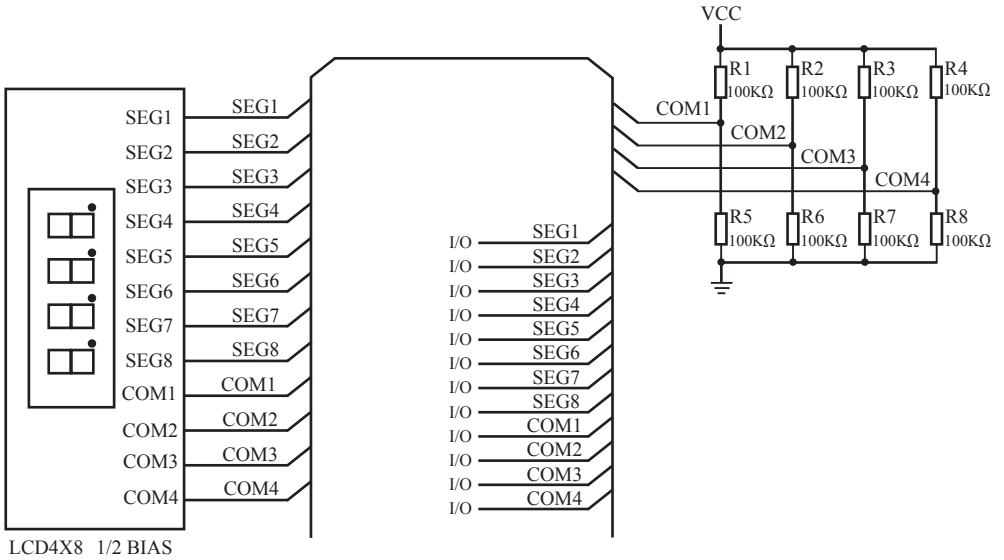


I/O dynamic scan driver 4 groups of digital tube Cathode circuit

I/O dynamic scan driver 4 groups of digital tube anode circuit



4.10 I/O immediately drive LCD application circuit



How to light on the LCD pixels:

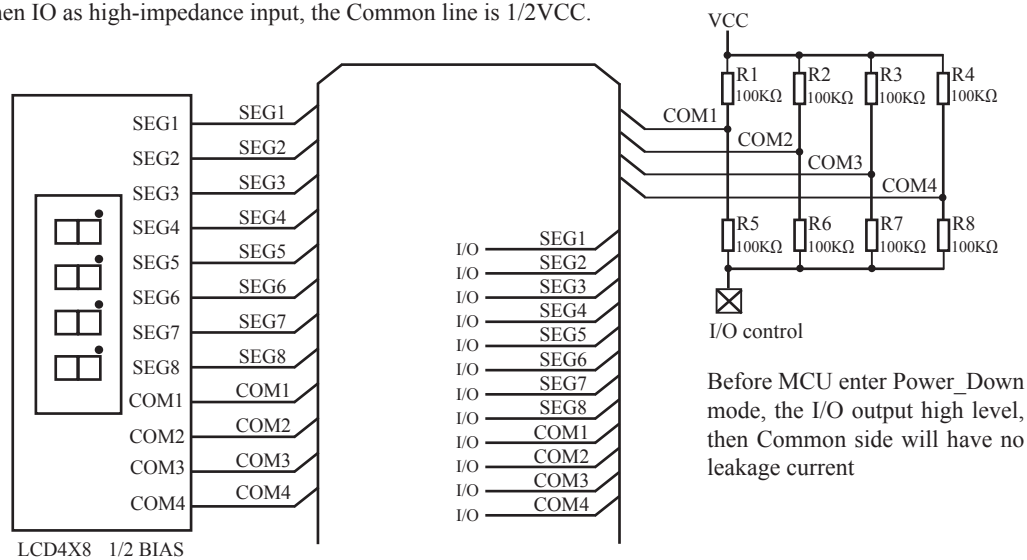
When the pixels corresponding COM-side and SEG-side voltage difference is greater than 1/2VCC, this pixel is lit, otherwise off

Contrl SEG-side (Segment) :

I/O direct drive Segment lines, control Segment output high-level (VCC) or low-level (0V).

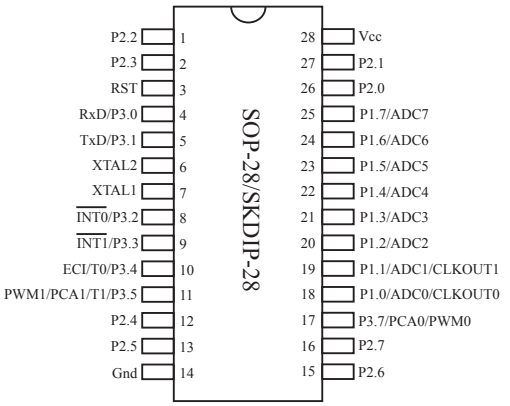
Contrl COM-side (Common) :

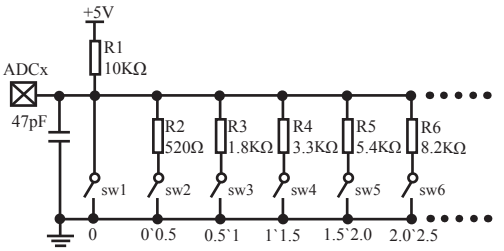
I/O port and two 100K dividing resistors jointly controlled Common line, when the IO output "0", the Common-line is low level (0V), when the IO push-pull output "1", the Common line is high level (VCC), when IO as high-impedance input, the Common line is 1/2VCC.



Before MCU enter Power_Down mode, the I/O output high level, then Common side will have no leakage current

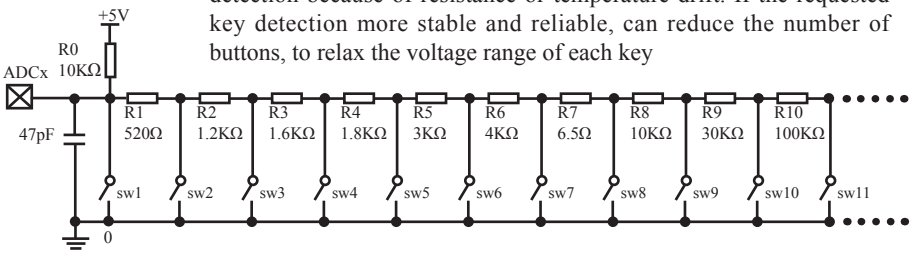
4.11 Using A/D Conversion to scan key application circuit





This circuit can achieve a signle key or combin key scan, resistance need to configure the actual needs

This circuit use 10 keys spaced partial pressure, for each key, range of allowed error is +/-0.25V, it can effectively avoid failure of key detection because of resistance or temperature drift. If the requested key detection more stable and reliable, can reduce the number of buttons, to relax the voltage range of each key



Chapter 5. Instruction System

5.1 Addressing Modes

Addressing modes are an integral part of each computer's instruction set. They allow specifyng the source or destination of data in different ways, depending on the programming situation. There are five modes available:

- Immediate
- Direct
- Indirect
- Register
- Indexed

Immediate Constant(IMM)

The value of a constant can follow the opcode in the program memory. For example,

```
MOV   A,  #70H
```

loads the Accumulator with the hex digits 70. The same number could be specified in decimal number as 112.

Direct Addressing(DIR)

In direct addressing the operand is specified by an 8-bit address field in the instruction. Only 128 lowest bytes of internal data RAM and SFRs can be direct addressed.

Indirect Addressing(IND)

In indirect addressing the instruction specified a register which contains the address of the operand. Both internal and external RAM can be indirectly addressed.

The address register for 8-bit addresses can be R0 or R1 of the selected bank, or the Stack Pointer.
The address register for 16-bit addresses can only be the 16-bit data pointer register – DPTR.

Register Instruction(REG)

The register banks, containing registers R0 through R7, can be accessed by certain instructions which carry a 3-bit register specification within the opcode of the instruction. Instructions that access the registers this way are code efficient because this mode eliminates the need of an extra address byte. When such instruction is executed, one of the eight registers in the selected bank is accessed.

Register-Specific Instruction

Some instructions are specific to a certain register. For example, some instructions always operate on the accumulator or data pointer,etc. No address byte is needed for such instructions. The opcode itself does it.

Index Addressing

Only program memory can be accessed with indexed addressing and it can only be read. This addressing mode is intended for reading look-up tables in program memory. A 16-bit base register(either DPTR or PC) points to the base of the table, and the accumulator is set up with the table entry number. Another type of indexed addressing is used in the conditional jump instruction.
In conditional jump, the destination address is computed as the sum of the base pointer and the accumulator.

5.2 Instruction Set Summary

The STC MCU instructions are fully compatible with the standard 8051's,which are divided among five functional groups:

- Arithmetic
- Logical
- Data transfer
- Boolean variable
- Program branching

The following tables provides a quick reference chart showing all the 8051 and STC 1T MCU instructions. Once you are familiar with the instruction set, this chart should prove a handy and quick source of reference.

Execution Clocks of Conventional 12T 8051

Execution Clocks of STC12C2052AD series

Mnemonic		Description	Byte	Execution clocks of 12T MCU	Execution clocks of STC 1T MCU	Efficiency improved
ARITHMETIC OPERATIONS						
ADD	A, Rn	Add register to Accumulator	1	12	2	6x
ADD	A, direct	Add ditect byte to Accumulator	2	12	3	4x
ADD	A, @Ri	Add indirect RAM to Accumulator	1	12	3	4x
ADD	A, #data	Add immediate data to Accumulator	2	12	2	6x
ADDC	A, Rn	Add register to Accumulator with Carry	1	12	2	6x
ADDC	A, direct	Add direct byte to Accumulator with Carry	2	12	3	4x
ADDC	A, @Ri	Add indirect RAM to Accumulator with Carry	1	12	3	4x
ADDC	A, #data	Add immediate data to Acc with Carry	2	12	2	6x
SUBB	A, Rn	Subtract Register from Acc wih borrow	1	12	2	6x
SUBB	A, direct	Subtract direct byte from Acc with borrow	2	12	3	4x
SUBB	A, @Ri	Subtract indirect RAM from ACC with borrow	1	12	3	4x
SUBB	A, #data	Substract immediate data from ACC with borrow	2	12	2	6x
INC	A	Increment Accumulator	1	12	2	6x
INC	Rn	Increment register	1	12	3	4x
INC	direct	Increment direct byte	2	12	4	3x
INC	@Ri	Increment direct RAM	1	12	4	3x
DEC	A	Decrement Accumulator	1	12	2	6x
DEC	Rn	Decrement Register	1	12	3	4x
DEC	direct	Decrement direct byte	2	12	4	3x
DEC	@Ri	Decrement indirect RAM	1	12	4	3x
INC	DPTR	Increment Data Pointer	1	24	1	24x
MUL	AB	Multiply A & B	1	48	4	12x
DIV	AB	Divde A by B	1	48	5	9.6x
DA	A	Decimal Adjust Accumulator	1	12	4	3x

Mnemonic		Description	Byte	Execution clocks of 12T MCU	Execution clocks of STC 1T MCU	Efficiency improved
LOGICAL OPERATIONS						
ANL	A, Rn	AND Register to Accumulator	1	12	2	6x
ANL	A, direct	AND direct btye to Accumulator	2	12	3	4x
ANL	A, @Ri	AND indirect RAM to Accumulator	1	12	3	4x
ANL	A, #data	AND immediate data to Accumulator	2	12	2	6x
ANL	direct, A	AND Accumulator to direct byte	2	12	4	3x
ANL	direct, #data	AND immediate data to direct byte	3	24	4	6x
ORL	A, Rn	OR register to Accumulator	1	12	2	6x
ORL	A,direct	OR direct byte to Accumulator	2	12	3	4x
ORL	A,@Ri	OR indirect RAM to Accumulator	1	12	3	4x
ORL	A, #data	OR immediate data to Accumulator	2	12	2	6x
ORL	direct, A	OR Accumulator to direct byte	2	12	4	3x
ORL	direct,#data	OR immediate data to direct byte	3	24	4	6x
XRL	A, Rn	Exclusive-OR register to Accumulator	1	12	2	6x
XRL	A, direct	Exclusive-OR direct byte to Accumulator	2	12	3	4x
XRL	A, @Ri	Exclusive-OR indirect RAM to Accumulator	1	12	3	4x
XRL	A, #data	Exclusive-OR immediate data to Accumulator	2	12	2	6x
XRL	direct, A	Exclusive-OR Accumulator to direct byte	2	12	4	3x
XRL	direct,#data	Exclusive-OR immediate data to direct byte	3	24	4	6x
CLR	A	Clear Accumulator	1	12	1	12x
CPL	A	Complement Accumulator	1	12	2	6x
RL	A	Rotate Accumulator Left	1	12	1	12x
RLC	A	Rotate Accumulator Left through the Carry	1	12	1	12x
RR	A	Rotate Accumulator Right	1	12	1	12x
RRC	A	Rotate Accumulator Right through the Carry	1	12	1	12x
SWAP	A	Swap nibbles within the Accumulator	1	12	1	12x

Mnemonic		Description	Byte	Execution clocks of 12T MCU	Execution clocks of STC 1T MCU	Efficiency improved
DATA TRANSFER						
MOV	A, Rn	Move register to Accumulator	1	12	1	12x
MOV	A, direct	Move direct byte to Accumulator	2	12	2	6x
MOV	A,@Ri	Move indirect RAM to	1	12	2	6x
MOV	A, #data	Move immediate data to Accumulator	2	12	2	6x
MOV	Rn, A	Move Accumulator to register	1	12	2	6x
MOV	Rn, direct	Move direct byte to register	2	24	4	6x
MOV	Rn, #data	Move immediate data to register	2	12	2	6x
MOV	direct, A	Move Accumulator to direct byte	2	12	3	4x
MOV	direct, Rn	Move register to direct byte	2	24	3	8x
MOV	direct,direct	Move direct byte to direct	3	24	4	6x
MOV	direct, @Ri	Move indirect RAM to direct byte	2	24	4	6x
MOV	direct, #data	Move immediate data to direct byte	3	24	3	8x
MOV	@Ri, A	Move Accumulator to indirect RAM	1	12	3	4x
MOV	@Ri, direct	Move direct byte to indirect RAM	2	24	4	6x
MOV	@Ri, #data	Move immediate data to indirect RAM	2	12	3	4x
MOV	DPTR, #data16	Move immdiate data to indirect RAM	2	24	3	8x
MOVC	A, @A+DPTR	Move Code byte relative to DPTR to Acc	1	24	4	6x
MOVC	A, @A+PC	Move Code byte relative to PC to Acc	1	24	4	6x
MOVX	A, @Ri	Move External RAM(8-bit addr) to Acc	1	24	3	8x
MOVX	@Ri, A	Move Acc to External RAM(8-bit addr)	1	24	4	6x
MOVX	A, @DPTR	Move External RAM(16-bit addr) to Acc	1	24	3	8x
MOVX	@DPTR, A	Move Acc to External RAM (16-bit addr)	1	24	3	8x
PUSH	direct	Push direct byte onto stack	2	24	4	6x
POP	direct	POP direct byte from stack	2	24	3	8x
XCH	A, Rn	Exchange register with Accumulator	1	12	3	4x
XCH	A, direct	Exchange direct byte with Accumulator	2	12	4	3x
XCH	A, @Ri	Exchange indirect RAM with Accumulator	1	12	4	3x
XCHD	A, @Ri	Exchange low-order Digit indirect RAM with Acc	1	12	4	3x

Mnemonic		Description	Byte	Execution clocks of 12T MCU	Execution clocks of STC 1T MCU	Efficiency improved
BOOLEAN VARIABLE MANIPULATION						
CLR	C	Clear Carry	1	12	1	12x
CLR	bit	Clear direct bit	2	12	4	3x
SETB	C	Set Carry	1	12	1	12x
SETB	bit	Set direct bit	2	12	4	3x
CPL	C	Complement Carry	1	12	1	12x
CPL	bit	Complement direct bit	2	12	4	3x
ANL	C, bit	AND direct bit to Carry	2	24	3	8x
ANL	C, /bit	AND complement of direct bit to Carry	2	24	3	8x
ORL	C, bit	OR direct bit to Carry	2	24	3	8x
ORL	C, /bit	OR complement of direct bit to Carry	2	24	3	8x
MOV	C, bit	Move direct bit to Carry	2	12	3	4x
MOV	bit, C	Move Carry to direct bit	2	24	4	6x
JC	rel	Jump if Carry is set	2	24	3	8x
JNC	rel	Jump if Carry not set	2	24	3	8x
JB	bit, rel	Jump if direct bit is set	3	24	4	6x
JNB	bit,rel	Jump if direct bit is not set	3	24	4	6x
JBC	bit, rel	Jump if direct bit is set & clear bit	3	24	5	4.8x
PROGRAM BRANCHING						
ACALL	addr11	Absolute Subroutine Call	2	24	6	4x
LCALL	addr16	Long Subroutine Call	3	24	6	4x
RET		Return from Subroutine	1	24	4	6x
RETI		Return from interrupt	1	24	4	6x
AJMP	addr11	Absolute Jump	2	24	3	8x
LJMP	addr16	Long Jump	3	24	4	6x
SJMP	rel	Short Jump (relative addr)	2	24	3	8x
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	24	3	8x
JZ	rel	Jump if Accumulator is Zero	2	24	3	8x
JNZ	rel	Jump if Accumulator is not Zero	2	24	3	8x
CJNE	A,direct,rel	Compare direct byte to Acc and jump if not equal	3	24	5	4.8x
CJNE	A,#data,rel	Compare immediate to Acc and Jump if not equal	3	24	4	6x
CJNE	Rn,#data,rel	Compare immediate to register and Jump if not equal	3	24	4	6x
CJNE	@Ri,#data,rel	Compare immediate to indirect and jump if not equal	3	24	5	4.8x
DJNZ	Rn, rel	Decrement register and jump if not Zero	2	24	4	6x
DJNZ	direct, rel	Decrement direct byte and Jump if not Zero	3	24	5	4.8x
NOP		No Operation	1	12	1	12x

Instruction execution speed boost summary:

24 times faster execution speed	1
12 times faster execution speed	12
9.6 times faster execution speed	1
8 times faster execution speed	20
6 times faster execution speed	39
4.8 times faster execution speed	4
4 times faster execution speed	20
3 times faster execution speed	14
24 times faster execution speed	1

Based on the analysis of frequency of use order statistics, STC 1T series MCU instruction execution speed is faster than the traditional 8051 MCU 8 ~ 12 times in the same working environment.

Instruction execution clock count:

1 clock instruction	12
2 clock instruction	20
3 clock instruction	38
4 clock instruction	34
5 clock instruction	5
6 clock instruction	2

5.3 Instruction Definitions

ACALL addr 11

Function: Absolute Call

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

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

ACALL SUBRTN

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

Bytes: 2

Cycles: 2

Encoding:

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

Operation: ACALL
(PC)← **(PC)+ 2**
(SP)←**(SP) + 1**
((sP)) ← **(PC_{7-0})**
(SP)←**(SP) + 1**
((SP))←**(PC_{15-8})**
(PC_{10-0})← **page address**

ADD A,<src-byte>

Function: Add

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

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

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

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

ADD A,R0

will leave 6DH (01101101B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A,Rn

Bytes: 1

Cycles: 1

Encoding:

0 0 1 0	1 r r r

Operation: ADD
(A)←(A) + (Rn)

ADD A,direct

Bytes: 2

Cycles: 1

Encoding:

0 0 1 0	0 1 0 1	direct address

Operation: ADD
(A)←(A) + (direct)

ADD A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 1 0	0 1 1 i

Operation: ADD
(A)←(A) + ((Ri))

ADD A,#data

Bytes: 2

Cycles: 1

Encoding:

0 0 1 0	0 1 0 0	immediate data

Operation: ADD
(A)←(A) + #data

ADDC A,<src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the Carry flag and the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occured.
OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.
Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

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

ADDC A,Rn

Bytes: 1

Cycles: 1

Encoding:

0 0 1 1	1 r r r

Operation: ADDC
(A)←(A) + (C) + (Rn)

ADDC A,direct

Bytes: 2

Cycles: 1

Encoding:

0 0 1 1	0 1 0 1	direct address

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

ADDC A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 1 1	0 1 1 i

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

ADDC A,#data

Bytes: 2

Cycles: 1

Encoding:

0 0 1 1	0 1 0 0	immediate data

Operation: ADDC
(A)←(A) + (C) + #data

AJMP addr 11

Function: Absolute Jump

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

Example: The label "JMPADR" is at program memory location 0123H. The instruction,

AJMP JMPADR

is at location 0345H and will load the PC with 0123H.

Bytes: 2

Cycles: 2

Encoding:

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

Operation: AJMP
(PC)← (PC)+ 2
($PC_{10\text{-}0}$)← page address

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

Function: Logical-AND for byte variables

Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.

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

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

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

ANL A,R0

will leave 41H (01000001B) in the Accumulator.

When the destination is a directly addressed byte, this instruction will clear combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the Accumulator at run-time. The instruction,

ANL Pl, #01110011B

will clear bits 7, 3, and 2 of output port 1.

ANL A,Rn

Bytes: 1

Cycles: 1

Encoding:

0 1 0 1	1 r r r

Operation: ANL
(A)←**(A)** ∧ (Rn)

ANL A,direct

Bytes: 2

Cycles: 1

Encoding:

0 1 0 1	0 1 0 1	direct address

Operation: ANL
(A)←**(A)** ∧ (direct)

ANL A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0 1 0 1	0 1 1 i

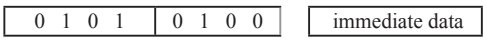
Operation: ANL
(A)←**(A)** ∧ ((Ri))

ANL A,#data

Bytes: 2

Cycles: 1

Encoding:



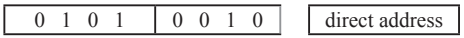
Operation: ANL
(A)←**(A)** ∧ #data

ANL direct,A

Bytes: 2

Cycles: 1

Encoding:



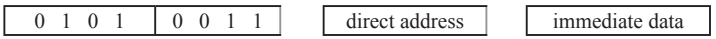
Operation: ANL
(direct)←**(direct)** ∧ (A)

ANL direct,#data

Bytes: 3

Cycles: 2

Encoding:



Operation: ANL
(direct)←**(direct)** ∧ #data

ANL C , <src-bit>

Function: Logical-AND for bit variables

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

Only direct addressing is allowed for the source operand.

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

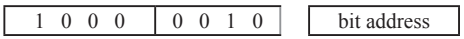
```
MOV   C, P1.0          ;LOAD CARRY WITH INPUT PIN STATE
ANL   C, ACC.7         ;AND CARRY WITH ACCUM. BIT.7
ANL   C, /OV           ;AND WITH INVERSE OF OVERFLOW FLAG
```

ANL C,bit

Bytes: 2

Cycles: 2

Encoding:



Operation: ANL
(C) ← **(C)** ∧ (bit)

ANL C, /bit

Bytes: 2

Cycles: 2

Encoding:

1 0 1 1	0 0 0 0	bit address

Operation: ADD

(C)←(C) ∧ $\overline{(bit)}$

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

Function: Compare and Jump if Not Equal

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

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

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

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

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

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

WAIT: CJNE A,P1,WAIT

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

CJNE A,direct,rel

Bytes: 3

Cycles: 2

Encoding:

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

Operation:

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

CJNE A,#data,rel

Bytes: 3

Cycles: 2

Encoding:

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

Operation:

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

CJNE Rn,#data,rel

Bytes: 3

Cycles: 2

Encoding:

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

Operation:

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

CJNE @Ri,#data,rel

Bytes: 3

Cycles: 2

Encoding:

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

Operation:

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

CLR A

Function: Clear Accumulator

Description: The Aecunmlator is cleared (all bits set on zero). No flags are affected.

Example: The Accumulator contains 5CH (01011100B). The instruction,

CLR A

will leave the Accumulator set to 00H (00000000B).

Bytes: 1

Cycles: 1

Encoding:

1 1 1 0	0 1 0 0

Operation: CLR

(A)← 0

CLR bit

Function: Clear bit

Description: The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.

Example: Port 1 has previously been written with 5DH (01011101B). The instruction,

CLR P1.2

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

CLR C

Bytes: 1

Cycles: 1

Encoding:

1 1 0 0	0 0 1 1

Operation: CLR

(C) ← 0

CLR bit

Bytes: 2

Cycles: 1

Encoding:

1 1 0 0	0 0 1 0	bit address

Operation: CLR

(bit) ← 0

CPL A

Function: Complement Accumulator

Description: Each bit of the Accumulator is logically complemented (one's complement). Bits which previously contained a one are changed to a zero and vice-versa. No flags are affected.

Example: The Accumulator contains 5CH(01011100B). The instruction,

```
CPL   A
```

will leave the Accumulator set to 0A3H (101000011B).

Bytes: 1

Cycles: 1

Encoding:

1 1 1 1	0 1 0 0

Operation: CPL

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

CPL bit

Function: Complement bit

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

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

Example: Port 1 has previously been written with 5DH (01011101B). The instruction,

```
CLR   P1.1
CLR   P1.2
```

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

CPL C

Bytes: 1

Cycles: 1

Encoding:

1 0 1 1	0 0 1 1

Operation: CPL

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

CPL bit

Bytes: 2

Cycles: 1

Encoding:

1 0 1 1	0 0 1 0	bit address

Operation: CPL

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

DA A

Function: Decimal-adjust Accumulator for Addition

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

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

If the carry flag is now set or if the four high-order bits now exceed nine(1010xxxx-111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this would set the carry flag if there was a carry-out of the high-order bits, but wouldn't clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected.

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

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

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

```
ADDC   A,R3
DA     A
```

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

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

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

```
ADD    A,#99H
DA     A
```

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

Bytes: 1

Cycles: 1

Encoding:

1 1 0 1	0 1 0 0

Operation: DA

-contents of Accumulator are BCD

IF [[(A_{3-0}) > 9] V [(AC) = 1]]

THEN(A_{3-0}) ← (A_{3-0}) + 6

AND

IF [[(A_{7-4}) > 9] V [(C) = 1]]

THEN (A_{7-4}) ← (A_{7-4}) + 6

DEC byte

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of 00H will underflow to 0FFH.

No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.

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

Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The instruction sequence,

```
DEC   @R0
DEC    R0
DEC   @R0
```

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

DEC A

Bytes: 1

Cycles: 1

Encoding:

0 0 0 1	0 1 0 0

Operation: DEC

(A)←(A) −1

DEC Rn

Bytes: 1

Cycles: 1

Encoding:

0 0 0 1	1 r r r

Operation: DEC

(Rn)←(Rn) - 1

DEC direct

Bytes: 2

Cycles: 1

Encoding:

0 0 0 1	0 1 0 1	direct address

Operation: DEC

(direct)←(direct) −1

DEC @Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 0 1	0 1 1 i

Operation: DEC

((Ri))←((Ri)) - 1

DIV AB

Function: Divide

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

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

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

DIV AB

will leave 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010010B) in B, since 251 = (13×18) + 17. Carry and OV will both be cleared.

Bytes: 1

Cycles: 4

Encoding:

1 0 0 0	0 1 0 0

Operation: DIV

$(A)_{15-8}$
$(B)_{7-0}$ ← (A)/(B)

DJNZ <byte>, <rel-addr>

Function: Decrement and Jump if Not Zero

Description: DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H will underflow to 0FFH. No flags are afected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

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

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

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

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

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

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

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

will toggle P1.7 eight times, causing four output pulses to appear at bit 7 of output Port 1. Each pulse will last three machine cycles; two for DJNZ and one to alter the pin.

DJNZ Rn,rel

Bytes: 2

Cycles: 2

Encoding:

1 1 0 1	1 r r r	rel. address

Operation: DJNZ

(PC) ← **(PC) + 2**

(Rn) ← **(Rn) – 1**

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

THEN

(PC) ← **(PC)+ rel**

DJNZ direct, rel

Bytes: 3

Cycles: 2

Encoding:

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

Operation: DJNZ
(PC) ← (PC) + 2
(direct) ← (direct) – 1
IF (direct) > 0 or (direct) < 0
THEN
(PC) ← (PC) + rel

INC <byte>

Function: Increment

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

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

Example: Register 0 contains 7EH (011111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The instruction sequence,

```
INC   @R0
INC    R0
INC   @R0
```

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

INC A

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	0 1 0 0

Operation: INC
(A) ← (A)+1

INC Rn

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	1 r r r

Operation: INC
(Rn) ← (Rn)+1

INC direct

Bytes: 2

Cycles: 1

Encoding:

0 0 0 0	0 1 0 1	direct address

Operation: INC
(direct)←(direct) + 1

INC @Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	0 1 1 i

Operation: INC
((Ri))←((Ri)) + 1

INC DPTR

Function: Increment Data Pointer

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

Example: Register DPH and DPL contains 12H and 0FEH,respectively. The instruction sequence,
INC DPTR
INC DPTR
INC DPTR
will change DPH and DPL to 13H and 01H.

Bytes: 1

Cycles: 2

Encoding:

1 0 1 0	0 0 1 1

Operation: INC
(DPTR) ← (DPTR)+1

JB bit, rel

Function: Jump if Bit set

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

Example: The data present at input port 1 is 11001010B. The Accumulator holds 56 (01010110B). The instruction sequence,
JB P1.2, LABEL1
JB ACC.2, LABEL2
will cause program execution to branch to the instruction at label LABEL2.

Bytes: 3

Cycles: 2

Encoding:

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

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

JBC bit, rel

Function: Jump if Bit is set and Clear bit

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

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

Example: The Accumulator holds 56H (01010110B). The instruction sequence,

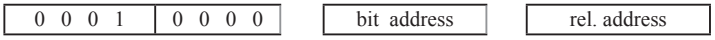
```
JBC     ACC.3, LABEL1
JBC     ACC.2, LABEL2
```

will cause program execution to continue at the instruction identified by the label LABEL2, with the Accumulator modified to 52H (01010010B).

Bytes: 3

Cycles: 2

Encoding:



Operation:

```
JBC
(PC) ← (PC)+ 3
IF  (bit) = 1
    THEN
        (bit) ← 0
        (PC) ← (PC) + rel
```

JC rel

Function: Jump if Carry is set

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

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

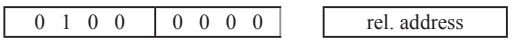
```
JC      LABEL1
CPL     C
JC      LABEL2s
```

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

Bytes: 2

Cycles: 2

Encoding:



Operation:

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

JMP @A+DPTR

Function: Jump indirect

Description: Add the eight-bit unsigned contents of the Accumulator with the sixteen-bit data pointer, and load the resulting sum to the program counter. This will be the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the Accumulator nor the Data Pointer is altered. No flags are affected.

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

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

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

Bytes: 1

Cycles: 2

Encoding:

0 1 1 1	0 0 1 1

Operation: JMP

(PC) ← (A) + (DPTR)

JNB bit, rel

Function: Jump if Bit is not set

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

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

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

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

Bytes: 3

Cycles: 2

Encoding:

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

Operation: JNB

(PC) ← (PC)+ 3

IF (bit) = 0

THEN (PC) ← (PC) + rel

JNC rel

Function: Jump if Carry not set

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

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

```
JNC   LABEL1
CPL   C
JNC   LABEL2
```

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

Bytes: 2

Cycles: 2

Encoding:

0 1 0 1	0 0 0 0	rel. address

Operation:

```
JNC
(PC) ← (PC)+ 2
IF  (C) = 0
    THEN    (PC) ← (PC) + rel
```

JNZ rel

Function: Jump if Accumulator Not Zero

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

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

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

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

Bytes: 2

Cycles: 2

Encoding:

0 1 1 1	0 0 0 0	rel. address

Operation:

```
JNZ
(PC) ← (PC)+ 2
IF  (A) ≠ 0
    THEN    (PC) ← (PC) + rel
```

JZ rel

Function: Jump if Accumulator Zero

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

Example: The Accumulator originally contains 01H. The instruction sequence,
JZ LABEL1
DEC A
JZ LAEEL2
will change the Accumulator to 00H and cause program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2

Cycles: 2

Encoding:

0 1 1 0	0 0 0 0	rel. address

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

LCALL addr16

Function: Long call

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

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

LCALL SUBRTN

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

Bytes: 3

Cycles: 2

Encoding:

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

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

LJMP addr16

Function: Long Jump

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

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

LJMP JMPADR

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

Bytes: 3

Cycles: 2

Encoding:

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

Operation: LJMP
(PC) ← $addr_{15-0}$

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

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

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

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

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

MOV A,Rn

Bytes: 1

Cycles: 1

Encoding:

1 1 1 0	1 r r r

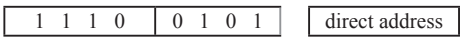
Operation: MOV
(A) ← (Rn)

***MOV A,direct**

Bytes: 2

Cycles: 1

Encoding:



Operation: MOV
(A)← (direct)

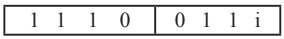
***MOV A, ACC is not a valid instruction**

MOV A,@Ri

Bytes: 1

Cycles: 1

Encoding:



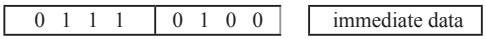
Operation: MOV
(A) ← ((Ri))

MOV A,#data

Bytes: 2

Cycles: 1

Encoding:



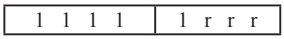
Operation: MOV
(A)← #data

MOV Rn, A

Bytes: 1

Cycles: 1

Encoding:



Operation: MOV
(Rn)←(A)

MOV Rn,direct

Bytes: 2

Cycles: 2

Encoding:

Operation: MOV
(Rn)←(direct)

MOV Rn,#data

Bytes: 2

Cycles: 1

Encoding:

Operation: MOV
(Rn) ← #data

MOV direct, A

Bytes: 2

Cycles: 1

Encoding:

1 1 1 1	0 1 0 1	direct address

Operation: MOV
(direct) ← (A)

MOV direct, Rn

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	1 r r r	direct address

Operation: MOV
(direct) ← (Rn)

MOV direct, direct

Bytes: 3

Cycles: 2

Encoding:

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

Operation: MOV
(direct)← (direct)

MOV direct, @Ri

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	0 1 1 i	direct addr.

Operation: MOV
(direct)←((Ri))

MOV direct,#data

Bytes: 3

Cycles: 2

Encoding:

0 1 1 1	0 1 0 1	direct address

Operation: MOV
(direct) ← #data

MOV @Ri, A

Bytes: 1

Cycles: 1

Encoding:

1 1 1 1	0 1 1 i

Operation: MOV
((Ri)) ← (A)

MOV @Ri, direct

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	0 1 1 i	direct addr.

Operation: MOV
((Ri)) ← (direct)

MOV @Ri, #data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	0 1 1 i	immediate data

Operation: MOV
((Ri)) ← #data

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

Function: Move bit data

Description: The Boolean variable indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example: The carry flag is originally set. The data present at input Port 3 is 11000101B. The data previously written to output Port 1 is 35H (00110101B).

```
MOV     P1.3, C
MOV     C, P3.3
MOV     P1.2, C
```

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

MOV C,bit

Bytes: 2

Cycles: 1

Encoding:

1 0 1 0	0 0 1 1	bit address

Operation: MOV
(C) ← (bit)

MOV bit,C

Bytes: 2

Cycles: 2

Encoding:

1 0 0 1	0 0 1 0	bit address

Operation: MOV
(bit)← (C)

MOV DPTR , #data 16

Function: Load Data Pointer with a 16-bit constant

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

Example: The instruction,
MOV DPTR, #1234H
will load the value 1234H into the Data Pointer: DPH will hold 12H and DPL will hold 34H.

Bytes: 3

Cycles: 2

Encoding:

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

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

MOVC A , @A+ <base-reg>

Function: Move Code byte

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

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

```
REL-PC:  INC    A
         MOVC   A, @A+PC
         RET
         DB     66H
         DB     77H
         DB     88H
         DB     99H
```

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

MOVC A,@A+DPTR

Bytes: 1

Cycles: 2

Encoding:

1 0 0 1	0 0 1 1

Operation: MOVC
(A) ← ((A)+(DPTR))

MOVC A,@A+PC

Bytes: 1

Cycles: 2

Encoding:

1 0 0 0	0 0 1 1

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

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

Function: Move External

Description: The MOVX instructions transfer data between the Accumulator and a byte of external data memory, hence the "X" appended to MOV. There are two types of instructions, differing in whether they provide an eight-bit or sixteen-bit indirect address to the external data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an eight-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX.

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

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

Example: An external 256 byte RAM using multiplexed address/data lines (e.g., an Intel 8155 RAM/I/O/Timer) is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence,

```
MOVX    A, @R1
MOVX    @R0, A
```

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

MOVX A,@Ri

Bytes: 1

Cycles: 2

Encoding:

1 1 1 0	0 0 1 i

Operation: MOVX
(A) ← ((Ri))

MOVX A,@DPTR

Bytes: 1

Cycles: 2

Encoding:

1 1 1 0	0 0 0 0

Operation: MOVX
(A) ← ((DPTR))

MOVX @Ri, A

Bytes: 1

Cycles: 2

Encoding:

1 1 1 1	0 0 1 i

Operation: MOVX
((Ri))← (A)

MOVX @DPTR, A

Bytes: 1

Cycles: 2

Encoding:

1 1 1 1	0 0 0 0

Operation: MOVX
(DPTR)←(A)

MUL AB

Function: Multiply

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

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

MUL AB

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

Bytes: 1

Cycles: 4

Encoding:

1 0 1 0	0 1 0 0

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

NOP

Function: No Operation

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

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

```
CLR     P2.7
NOP
NOP
NOP
NOP
SETB    P2.7
```

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	0 0 0 0

Operation: NOP
(PC) ← (PC)+1

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

Function: Logical-OR for byte variables

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

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

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

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

```
ORL     A, R0
```

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

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

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

ORL A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	1	0	0	1	r	r	r

Operation: ORL
(A) ← **(A)**∨(Rn)

ORL A,direct

Bytes: 2

Cycles: 1

Encoding:

0	1	0	0	0	1	0	1	direct address

Operation: ORL
(A)← **(A)**∨(direct)

ORL A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0	1	0	0	0	1	1	i

Operation: ORL
(A)← **(A)**∨((Ri))

ORL A,#data

Bytes: 2

Cycles: 1

Encoding:

0	1	0	0	0	1	0	0	immediate data

Operation: ORL
(A)← **(A)**∨ #data

ORL direct, A

Bytes: 2

Cycles: 1

Encoding:

0	1	0	0	0	0	1	0	direct address

Operation: ORL
(direct)← **(direct)**∨(A)

ORL direct, #data

Bytes: 3

Cycles: 2

Encoding:

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

Operation: ORL
(direct) ← **(direct)**∨#data

ORL C, <src-bit>

Function: Logical-OR for bit variables

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

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

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

ORL C, bit

Bytes: 2

Cycles: 2

Encoding:

0 1 1 1	0 0 1 0	bit address

Operation: ORL

(C) ← (C)∨(bit)

ORL C, /bit

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	0 0 0 0	bit address

Operation: ORL

(C) ← (C)∨$\overline{(bit)}$

POP direct

Function: Pop from stack

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

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

```
POP     DPH
POP     DPL
```

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

```
POP     SP
```

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

Bytes: 2

Cycles: 2

Encoding:

1 1 0 1	0 0 0 0	direct address

Operation: POP

(diect) ← ((SP))

(SP) ← (SP) - 1

PUSH direct

Function: Push onto stack

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

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

```
PUSH    DPL
PUSH    DPH
```

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

Bytes: 2

Cycles: 2

Encoding:

1 1 0 0	0 0 0 0	direct address

Operation: PUSH
(SP) ← (SP) + 1
((SP)) ← (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}) ← ((SP))
(SP) ← (SP) -1
(PC_{7-0}) ← ((SP))
(SP) ← (SP) -1

RETI

Function: Return from interrupt

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

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

RETI

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

Bytes: 1

Cycles: 2

Encoding:

0	0	1	1	0	0	1	0

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

RL A

Function: Rotate Accumulator Left

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

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

RL A

leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	0	0	1	1

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

RLC A

Function: Rotate Accumulator Left through the Carry flag

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

Example: The Accumulator holds the value 0C5H (11000101B), and the carry is zero. The instruction,
RLC A
leaves the Accumulator holding the value 8BH (10001011B) with the carry set.

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	0	0	1	1

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

RR A

Function: Rotate Accumulator Right

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

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

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	1	1

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

RRC A

Function: Rotate Accumulator Right through the Carry flag

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

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

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	0	1	1

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

SETB <bit>

Function: Set bit

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

Example: The carry flag is cleared. Output Port 1 has been written with the value 34H (00110100B). The instructions,

```
SETB    C
SETB    P1.0
```

will leave the carry flag set to 1 and change the data output on Port 1 to 35H (00110101B).

SETB C

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	0	1	1

Operation: SETB

(C) ← 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.

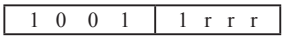
Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the carry (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.

SUBB A, Rn

Bytes: 1

Cycles: 1

Encoding:



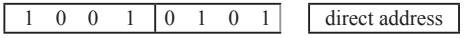
Operation: SUBB
(A) ← (A) - (C) - (Rn)

SUBB A, direct

Bytes: 2

Cycles: 1

Encoding:



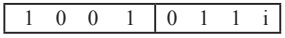
Operation: SUBB
(A) ← (A) - (C) - (direct)

SUBB A, @Ri

Bytes: 1

Cycles: 1

Encoding:



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

SUBB A, #data

Bytes: 2

Cycles: 1

Encoding:

1 0 0 1	0 1 0 0	immediate data

Operation: SUBB

(A) ← (A) - (C) - #data

SWAP A

Function: Swap nibbles within the Accumulator

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

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

SWAP A

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

Bytes: 1

Cycles: 1

Encoding:

1 1 0 0	0 1 0 0

Operation: SWAP

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

XCH A, <byte>

Function: Exchange Accumulator with byte variable

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

Example: R0 contains the address 20H. The Accumulator holds the value 3FH (00111111B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

XCH A, @R0

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

XCH A, Rn

Bytes: 1

Cycles: 1

Encoding:

1 1 0 0	1 r r r

Operation: XCH

(A) ⇄ (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) $\rightleftarrows$ ((Ri))

XCHD A, @Ri

Function: Exchange Digit

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

Example: R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

XCHD A, @R0

will leave RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the accumulator.

Bytes: 1

Cycles: 1

Encoding:

1 1 0 1	0 1 1 i

Operation: XCHD

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

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

Function: Logical Exclusive-OR for byte variables

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

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

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

Example: If the Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) then the instruction,

XRL A, R0

will leave the Accumulator holding the vatue 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combinnation of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the Accumulator at run-time. The instruction,

XRL P1, #00110001B

will complement bits 5,4 and 0 of outpue Port 1.

XRL A, Rn

Bytes: 1

Cycles: 1

Encoding:

0 1 1 0	1 r r r

Operation: XRL
(A) ← (A) ⩛ (Rn)

XRL A, direct

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 1 0 1	direct address

Operation: XRL
(A) ← (A) ⩛ (direct)

XRL A, @Ri

Bytes: 1

Cycles: 1

Encoding:

0 1 1 0	0 1 1 i

Operation: XRL
(A) ← (A) ⩛ ((Ri))

XRL A, #data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 1 0 0	immediate data

Operation: XRL
(A) ← (A) ⩛ #data

XRL direct, A

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 0 1 0	direct address

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

XRL direct, #dataw

Bytes: 3

Cycles: 2

Encoding:

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

Operation: XRL
(direct) ← (direct) ⩛ # data

Chapter 6. Interrupt System

STC12C2052AD series support 9 interrupt sources with four priority levels. The 9 interrupt sources are external interrupt 0($\overline{\text{INT0}}$), Timer 0 interrrupt, external interrupt 1($\overline{\text{INT1}}$), Timer 1 interrrupt, serial port 1(UART) interrupt, ADC and SPI interrupt, low voltage detector (LVD) and PCA interrupt. Each interrupt source has one or more associated interrupt-request flag(s) in SFRs. Associating with each interrupt vector, the interrupt sources can be individually enabled or disabled by setting or clearing a bit (interrupt enalbe control bit) in the SFRs IE, CCON. However, interrupts must first be globally enabled by setting the EA bit (IE.7) to logic 1 before the individual interrupt enables are recognized. Setting the EA bit to logic 0 disables all interrupt sources regardless of the individual interrupt-enable settings.

If interrupts are enabled for the source, an interrupt request is generated when the interrupt-request flag is set. As soon as execution of the current instruction is complete, the CPU generates an LCALL to a predetermined address to begin execution of an interrupt service routine (ISR). Each ISR must end with an RETI instruction, which returns program execution to the next instruction that would have been executed if the interrupt request had not occurred. If interrupts are not enabled, the interrupt-pending flag is ignored by the hardware and program execution continues as normal. (The interrupt-pending flag is set to logic 1 regardless of the interrupt's enable/disable state.)

Each interrupt source has two corresponding bits to represent its priority. One is located in SFR named IPH and other in IP register. Higher-priority interrupt will be not interrupted by lower-priority interrupt request. If two interrupt requests of different priority levels are received simultaneously, the request of higher priority is serviced. If interrupt requests of the same priority level are received simultaneously, an internal polling sequence determine which request is serviced. The following table shows the internal polling sequence in the same priority level and the interrupt vector address.

Interrupt Sources, vector address, priority and polling sequence Table

Interrupt Source	Interrupt Vector address	Priority within level	Interrupt Priority setting(IPH, IP)	Priority 0 (lowest)	Priority 1	Priority 2	Priority 3 (highest)	Interrupt Request	Interrupt Enable Control Bit
External interrupt 0 ($\overline{\text{INT0}}$)	0003H	0(highest)	PX0H,PX0	0,0	0,1	1,0	1,1	IE0	EX0/EA
Timer 0	000BH	1	PT0H,PT0	0,0	0,1	1,0	1,1	TF0	ET0/EA
External interrupt 1 ($\overline{\text{INT1}}$)	0013H	2	PX1H,PX1	0,0	0,1	1,0	1,1	IE1	EX1/EA
Timer1	001BH	3	PT1H, PT1	0, 0	0, 1	1, 0	1, 1	TF1	ET1/EA
UART	0023H	4	PSH, PS	0, 0	0, 1	1, 0	1, 1	RI+TI	ES/EA
ADC/SPI	002BH	5	PADC_SPIH, PADC_SPI	0, 0	0, 1	1, 0	1, 1	ADC_FLAG +SPIF	(EADCI+ESPI) / EADC_SPI / EA
PCA/ LVD	0033H	6 (lowest)	PPCA_LVDH, PPCA_LVD	0, 0	0, 1	1, 0	1, 1	CF+CCF0+ CCF1++LVDF	(ECF+ECCF0+ECCF1 +ELVD) / EPCA_LVD / EA

In C language program. the interrupt polling sequence number is equal to interrupt number, for example,

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

6.1 Interrupt Structure

The interrupt structure of STC12C2052AD series is shown as below.

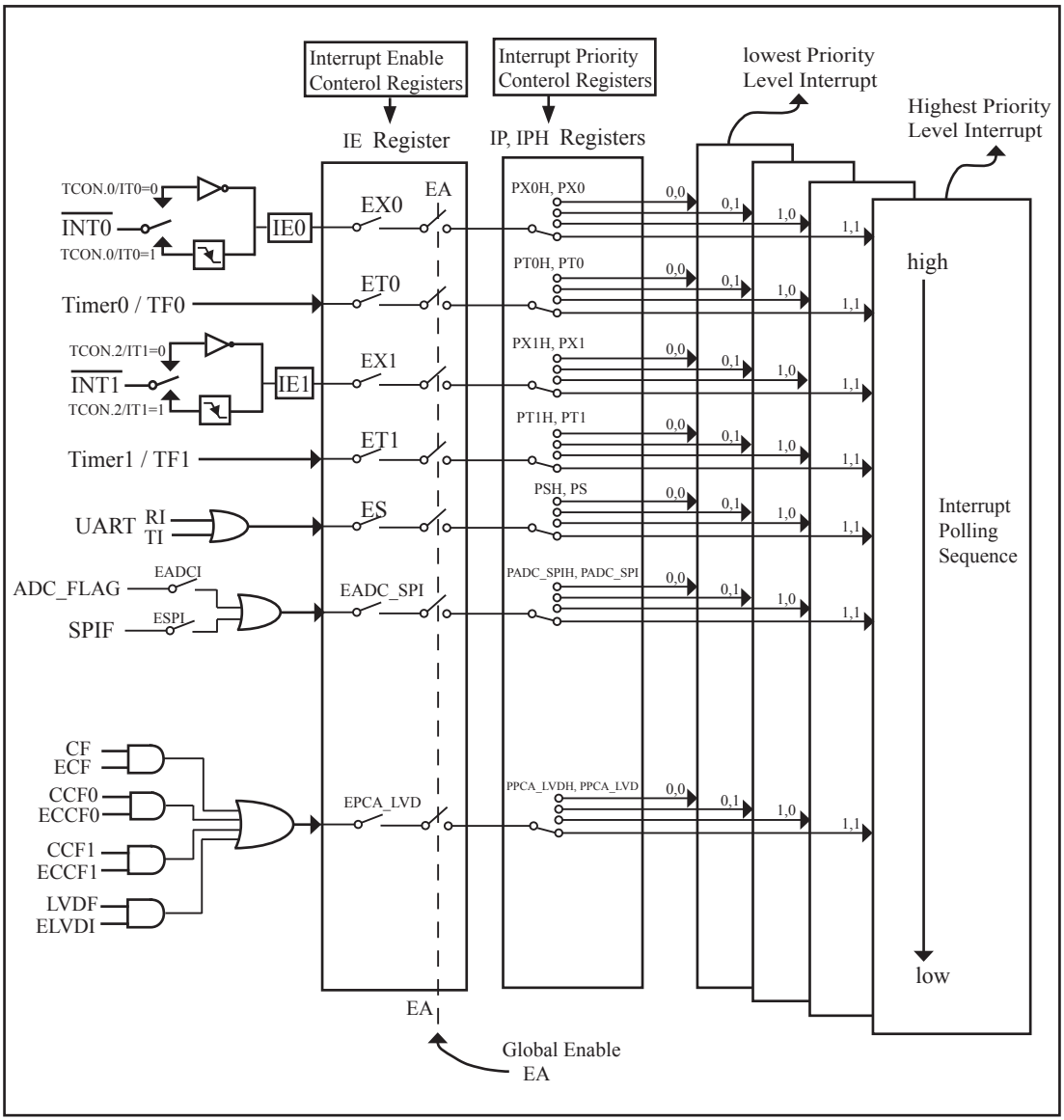


Figure STC12C2052AD series Interrupt Structure diagram

The External Interrupts $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ can each be either level-activated or transition-activated, depending on bits IT0 and IT1 in Register TCON. The flags that actually generate these interrupts are bits IE0 and IE1 in TCON. When an external interrupt is generated, the flag that generated it is cleared by the hardware when the service routine is vectored to if and only if the interrupt was transition –activated, otherwise the external requesting source is what controls the request flag, rather than the on-chip hardware.

The Timer 0 and Timer1 Interrupts are generated by TF0 and TF1, which are set by a rollover in their respective Timer/Counter registers in most cases. When a timer interrupt is generated, the flag that generated it is cleared by the on-chip hardware when the service routine is vectored to.

The Serial Port Interrupt is generated by the logical OR of RI and TI. Neither of these flags is cleared by hardware when the service routine is vectored to. In fact, the service routine will normally have to determine whether it was RI and TI that generated the interrupt, and the bit will have to be cleared by software.

The 2BH interrupt is shared by the logical “1” of SPI interrupt and ADC interrupt.Neither of these flags is cleared by hardware when the service routine is vectored to. The service routine should poll them to determine which one to request service and it will be cleared by software.

The 33H interrupt is shared by the logical “1” of PCA interrupt and LVD (Low-Voltage Detector) interrupt. Neither of these flags is cleared by hardware when the service routine is vectored to. The service routine should poll them to determine which one to request service and it will be cleared by software.

All of the bits that generate interrupts can be set or cleared by software, with the same result as though it had been set or cleared by hardware. In other words, interrupts can be generated or pending interrupts can be canceled in software.

6.2 Interrupt Register

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
IE	Interrupt Enable	A8H	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0	0000 0000B
IP	Interrupt Priority Low	B8H	-	PPCA_LVD	PADC_SPI	PS	PT1	PX1	PT0	PX0	x000 0000B
IPH	Interrupt Priority High	B7H	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H	x000 0000B
TCON	Timer Control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
SCON	Serial Control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000 0000B
AUXR	Auxiliary register	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
PCON	Power Control	87H	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011 0000B
ADC_CONTR	ADC Control	C5H	ADC_POWER	SPEED1	SPEED0	ADC_FLAG	ADC_START	CHS2	CHS1	CHS0	0000 0000B
CCON	PCA Control Register	D8H	CF	CR	-	-	-	-	CCF1	CCF0	00xx xx00B
CMOD	PCA Mode Register	D9H	CIDL	-	-	-	-	CPS1	CPS0	ECF	00xx 0000B
CCAPM0	PCA Module 0 Mode Register	DAH	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0	x000 0000B
CCAPM1	PCA Module 1 Mode Register	DBH	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1	x000 0000B
SPSTAT	SPI Status register	84H	SPIF	WCOL	-	-	-	-	-	-	

1. Interrupt Enable control Register IE and AUXR

IE: Interrupt Enable Rsgister (Bit-addressable)

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

Enable Bit = 1 enables the interrupt .
Enable Bit = 0 disables it .

EA (IE.7): disables all interrupts. If EA = 0,no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

EPCA_LVD (IE.6): Interrupt controller of Programmable Counter Array (PCA) and Low-Voltage Detector.

0 : Disable the interrupt of Programmable Counter Array (PCA) and Low-Voltage Detector

1 : Enable the interrupt of Programmable Counter Array (PCA) and Low-Voltage Detector

EADC_SPI (IE.5): Interrupt controller of Serial Peripheral Interface (SPI) and A/D Converter (ADC).

0 : Disable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).

1 : Enable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).

ES (IE.4): Serial Port 1 (UART1) interrupt enable bit. If ES = 0, UART1 interrupt will be diabled. If ES = 1, UART1 interrupt is enabled.

ET1 (IE.3): Timer 1 interrupt enable bit. If ET1 = 0, Timer 1 interrupt will be diabled. If ET1 = 1, Timer 1 interrupt is enabled.

EX1 (IE.2): External interrupt 1 enable bit. If EX1 = 0, external interrupt 1 will be diabled. If EX1 = 1, external interrupt 1 is enabled.

ET0 (IE.1): Timer 0 interrupt enable bit. If ET0 = 0, Timer 0 interrupt will be diabled. If ET0 = 1, Timer 0 interrupt is enabled.

EX0 (IE.0): External interrupt 0 enable bit. If EX0 = 0, external interrupt 0 will be diabled. If EX0 = 1, external interrupt 0 is enabled.

AUXR: Auxiliary register (Non bit-addressable)

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

ELVDI : Enable/Disable interrupt from low-voltage sensor

0 : (default)Inhibit the low-voltage sensor functional block to generate interrupt to the MCU

1 : Enable the low-voltage sensor functional block to generate interrupt to the MCU

ESPI : Enable/Disable interrupt from Serial Peripheral Interface (SPI)

0 : (default)Inhibit the SPI functional block to generate interrupt to the MCU

1 : Enable the SPI functional block to generate interrupt to the MCU

EADCI : Enable/Disable interrupt from A/D converter

0 : (default) Inhibit the ADC functional block to generate interrupt to the MCU

1 : Enable the ADC functional block to generate interrupt to the MCU

UART_M0x6 : Baud rate select bit of UART1 while it is working under Mode-0

0 : The baud-rate of UART in mode 0 is SYSclk/12.

1 : The baud-rate of UART in mode 0 is SYSclk/2.

T1x12 : Timer 1 clock source bit.

0 : The clock source of Timer 1 is SYSclk/12. It will compatible to the traditional 80C51 MCU

1 : The clock source of Timer 1 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

T0x12 : Timer 0 clock source bit.

0 : The clock source of Timer 0 is SYSclk/12. It will compatible to the traditional 80C51 MCU

1 : The clock source of Timer 0 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

2. Interrupt Priority control Registers IP and IPH

Each interrupt source of STC12C2052AD all can be individually programmed to one of four priority levels by setting or clearing the bits in Special Function Registers IP and IPH. A low-priority interrupt can itself be interrupted by a high-pority interrupt, but not by another low-priority interrupt. A high-priority interrupt can't be interrupted by any other interrupt source.

IPH: Interrupt Priority High Register (Non bit-addressable)

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

IP: Interrupt Priority Register (Bit-addressable)

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

PPCA_LVDH, PPCA_LVD: Programmable Counter Array (PCA) and Low voltage detector interrupt priority control bits.

if PPCA_LVDH=0 and PPCA_LVD=0, Programmable Counter Array (PCA) and Low voltage detector interrupt are assigned lowest priority(priority 0).

if PPCA_LVDH=0 and PPCA_LVD=1, Programmable Counter Array (PCA) and Low voltage detector interrupt are assigned lowest priority(priority 1).

if PPCA_LVDH=1 and PPCA_LVD=0, Programmable Counter Array (PCA) and Low voltage detector interrupt are assigned lowest priority(priority 2).

if PPCA_LVDH=1 and PPCA_LVD=1, Programmable Counter Array (PCA) and Low voltage .detector interrupt are assigned lowest priority(priority 3).

PADC_SPIH, PADC_SPI : Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt priority control bits.

if PADC_SPIH=0 and PADC_SPI=0, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 0).

if PADC_SPIH=0 and PADC_SPI=1, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 1).

if PADC_SPIH=1 and PADC_SPI=0, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 2).

if PADC_SPIH=1 and PADC_SPI=1, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 3).

PSH, PS: Serial Port 1 (UART1) interrupt priority control bits.
if PSH=0 and PS=0, UART1 interrupt is assigned lowest priority (priority 0).
if PSH=0 and PS=1, UART1 interrupt is assigned lower priority (priority 1).
if PSH=1 and PS=0, UART1 interrupt is assigned higher priority (priority 2).
if PSH=1 and PS=1, UART1 interrupt is assigned highest priority (priority 3).

PT1H, PT1: Timer 1 interrupt priority control bits.
if PT1H=0 and PT1=0, Timer 1 interrupt is assigned lowest priority (priority 0).
if PT1H=0 and PT1=1, Timer 1 interrupt is assigned lower priority (priority 1).
if PT1H=1 and PT1=0, Timer 1 interrupt is assigned higher priority (priority 2).
if PT1H=1 and PT1=1, Timer 1 interrupt is assigned highest priority (priority 3).

PX1H, PX1: External interrupt 1 priority control bits.
if PX1H=0 and PX1=0, External interrupt 1 is assigned lowest priority (priority 0).
if PX1H=0 and PX1=1, External interrupt 1 is assigned lower priority (priority 1).
if PX1H=1 and PX1=0, External interrupt 1 is assigned higher priority (priority 2).
if PX1H=1 and PX1=1, External interrupt 1 is assigned highest priority (priority 3).

PT0H, PT0: Timer 0 interrupt priority control bits.
if PT0H=0 and PT0=0, Timer 0 interrupt is assigned lowest priority (priority 0).
if PT0H=0 and PT0=1, Timer 0 interrupt is assigned lower priority (priority 1).
if PT0H=1 and PT0=0, Timer 0 interrupt is assigned higher priority (priority 2).
if PT0H=1 and PT0=1, Timer 0 interrupt is assigned highest priority (priority 3).

PX0H, PX0: External interrupt 0 priority control bits.
if PX0H=0 and PX0=0, External interrupt 0 is assigned lowest priority (priority 0).
if PX0H=0 and PX0=1, External interrupt 0 is assigned lower priority (priority 1).
if PX0H=1 and PX0=0, External interrupt 0 is assigned higher priority (priority 2).
if PX0H=1 and PX0=1, External interrupt 0 is assigned highest priority (priority 3).

3. TCON register: Timer/Counter Control Register (Bit-Addressable)

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

TF1: Timer/Counter 1 Overflow Flag. Set by hardware on Timer/Counter 1 overflow. The flag can be cleared by software but is automatically cleared by hardware when processor vectors to the Timer 1 interrupt routine.
If TF1 = 0, No Timer 1 overflow detected.
If TF1 = 1, Timer 1 has overflowed.

TR1: Timer/Counter 1 Run Control bit. Set/cleared by software to turn Timer/Counter on/off.
If TR1 = 0, Timer 1 disabled.
If TR1 = 1, Timer 1 enabled.

TF0: Timer/Counter 0 Overflow Flag. Set by hardware on Timer/Counter 0 overflow. The flag can be cleared by software but is automatically cleared by hardware when processor vectors to the Timer 0 interrupt routine.
If TF0 = 0, No Timer 0 overflow detected.
If TF0 = 1, Timer 0 has overflowed.

TR0: Timer/Counter 0 Run Control bit. Set/cleared by software to turn Timer/Counter on/off.
If TR0 = 0, Timer 0 disabled.
If TR0 = 1, Timer 0 enabled.

IE1: External Interrupt 1 Edge flag. Set by hardware when external interrupt edge/level defined by IT1 is detected. The flag can be cleared by software but is automatically cleared when the external interrupt 1 service routine has been processed.

IT1: External Intenupt 1 Type Select bit. Set/cleared by software to specify falling edge/low level triggered external interrupt 1.
If IT1 = 0, $\overline{\text{INT1}}$ is low level triggered.
If IT1 = 1, $\overline{\text{INT1}}$ is edge triggered.

IE0: External Interrupt 0 Edge flag. Set by hardware when external interrupt edge/level defined by IT0 is detected. The flag can be cleared by software but is automatically cleared when the external interrupt 0 service routine has been processed.

IT0: External Intenupt 0 Type Select bit. Set/cleared by software to specify falling edge/low level triggered external interrupt 0.
If IT0 = 0, $\overline{\text{INT0}}$ is low level triggered.
If IT0 = 1, $\overline{\text{INT0}}$ is edge triggered.

4. SCON register: Serial Port 1 (UART1) Control Register (Bit-Addressable)

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

FE: Framing Error bit. The SMOD0 bit must be set to enable access to the FE bit

0: The FE bit is not cleared by valid frames but should be cleared by software.

1: This bit set by the receiver when an invalid stop bit id detected.

SM0,SM1 : Serial Port Mode Bit 0/1.

SM0	SM1	Description	Baud rate
0	0	8-bit shift register	SYSclk/12
0	1	8-bit UART	variable
1	0	9-bit UART	SYSclk/64 or SYSclk/32(SMOD=1)
1	1	9-bit UART	variable

SM2 : Enable the automatic address recognition feature in mode 2 and 3. If SM2=1, RI will not be set unless the received 9th data bit is 1, indicating an address, and the received byte is a Given or Broadcast address. In mode1, if SM2=1 then RI will not be set unless a valid stop Bit was received, and the received byte is a Given or Broadcast address. In mode 0, SM2 should be 0.

REN : When set enables serial reception.

TB8 : The 9th data bit which will be transmitted in mode 2 and 3.

RB8 : In mode 2 and 3, the received 9th data bit will go into this bit.

TI : Transmit interrupt flag. Set by hardware when a byte of data has been transmitted by UART0 (after the 8th bit in 8-bit UART Mode, or at the beginning of the STOP bit in 9-bit UART Mode). When the UART0 interrupt is enabled, setting this bit causes the CPU to vector to the UART0 interrupt service routine. This bit must be cleared manually by software.

RI : Receive interrupt flag. Set to ‘1’ by hardware when a byte of data has been received by UART0 (set at the STOP bit sam-pling time). When the UART0 interrupt is enabled, setting this bit to ‘1’ causes the CPU to vector to the UART0 interrupt service routine. This bit must be cleared manually by software.

5. Register related with LVD interrupt: Power Control register PCON (Non bit-Addressable)

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

SMOD: double Baud rate control bit.

0 : Disable double Baud rate of the UART.

1 : Enable double Baud rate of the UART in mode 1,2,or 3.

SMOD0: Frame Error select.

0 : SCON.7 is SM0 function.

1 : SCON.7 is FE function. Note that FE will be set after a frame error regardless of the state of SMOD0.

LVDF : Pin Low-Voltage Flag. Once low voltage condition is detected (VCC power is lower than LVD voltage), it is set by hardware (and should be cleared by software).

POF : Power-On flag. It is set by power-off-on action and can only cleared by software.

GF1 : General-purposed flag 1

GF0 : General-purposed flag 0

PD : Power-Down bit.

IDL : Idle mode bit.

IE: Interrupt Enable Rsgister (Bit-addressable)

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

EA : disables all interrupts.

If EA = 0,no interrupt will be acknowledged.

If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

EPCA_LVD : Interrupt controller of Programmable Counter Array (PCA) and Low-Voltage Detector.

If EPCA_LVD = 0, Disable the interrupt of Programmable Counter Array (PCA) and Low-Voltage Detector.

If EPCA_LVD = 1, Enable the interrupt of Programmable Counter Array (PCA) and Low-Voltage Detector

6. ADC_CONTR: AD Control register (Non bit-Addressable)

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

ADC_POWER : When clear, shut down the power of ADC bolck. When set, turn on the power of ADC block.

ADC_FLAG : ADC interrupt flag.It will be set by the device after the device has finished a conversion, and should be cleared by the user's software.

ADC_STRAT : ADC start bit, which enable ADC conversion.It will automatically cleared by the device after the device has finished the conversion

IE: Interrupt Enable Rsgister (Bit-addressable)

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

EA : disables all interrupts.
If EA = 0,no interrupt will be acknowledged.
If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

EADC_SPI : Interrupt controller of Serial Peripheral Interface (SPI) and A/D Converter (ADC).
If EADC_SPI = 0, Disable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).
If EADC_SPI = 1, Enable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).

7. Register related with PCA interrupt

CCON: PCA Control Register (bit-Addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CCON	D8H	name	CF	CR	-	-	-	-	CCF1	CCF0

CF : PCA Counter Overflow flag. Set by hardware when the counter rolls over. CF flags an interrupt if bit ECF in CMOD is set. CF may be set by either hardware or software but can only be cleared by software.

CR : PCA Counter Run control bit. Set by software to turn the PCA counter on. Must be cleared by software to turn the PCA counter off.

CCF1: PCA Module 1 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software.

CCF0: PCA Module 0 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software.

CMOD: PCA Mode Register (Non bit-Addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CMOD	D9H	name	CIDL	-	-	-	-	CPS1	CPS0	ECF

CIDL : Counter Idle control. CIDL=0 programs the PCA Counter to continue functioning during idle mode. CIDL=1 programs it to be gated off during idle.

CPS1 ~ CPS0 : PCA Counter Pulse Select bits.

0	0	Internal clock, fosc/12
0	1	Internal clock, fosc/2
0	0	Timer 0 overflow
0	1	External clock at ECI/P3.4 pin

ECF : PCA Enable Counter Overflow interrupt. ECF=1 enables CF bit in CCON to generate an interrupt.

CCAPMn register (Non bit-Addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CCAPM0	DAH	name	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0
CCAPM1	DBH	name	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1

ECOMn : Enable Comparator. ECOMn=1 enables the comparator function.

CAPPn : Capture Positive, CAPPn=1 enables positive edge capture.

CAPNn : Capture Negative, CAPNn=1 enables negative edge capture.

MATn : Match. When MATn=1, a match of the PCA counter with this module's compare/capture register causes the CCFn bit in CCON to be set.

TOGn : Toggle. When TOGn=1, a match of the PCA counter with this module's compare/capture register causes the CEXn pin to toggle.

PWMn : Pulse Width Modulation. PWMn=1 enables the CEXn pin to be used as a pulse width modulated output.

ECCFn : Enable CCF interrupt. Enables compare/capture flag CCFn in the CCON register to generate

IE: Interrupt Enable Rsgister (Bit-addressable)

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

EA : disables all interrupts.

If EA = 0,no interrupt will be acknowledged.

If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

EPCA_LVD : Interrupt controller of Programmable Counter Array (PCA) and Low-Voltage Detector.

If EPCA_LVD = 0, Disable the interrupt of Programmable Counter Array (PCA) and Low-Voltage Detector.

If EPCA_LVD = 1, Enable the interrupt of Programmable Counter Array (PCA) and Low-Voltage Detector

8. Register related with SPI interrupt : SPSTAT

SPI State register: SPSTAT (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPSTAT	CDH	name	SPIF	WCOL	-	-	-	-	-	-

SPIF : SPI transfer completion flag.
When a serial transfer finishes, the SPIF bit is set and an interrupt is generated if all the EADC_SPI (IE.6) bit, ESPI bit (AUXR.3) and the EA (IE.7) bit are set. If SS is an input and is driven low when SPI is in master mode with SSIG = 0, SPIF will also be set to signal the "mode change".The SPIF is cleared in software by "writing 1 to this bit".

WCOL: SPI write collision flag.
The WCOL bit is set if the SPI data register, SPDAT, is written during a data transfer. The WCOL flag is cleared in software by "writing 1 to this bit".

IE: Interrupt Enable Rsgister (Bit-addressable)

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

EA : disables all interrupts.
If EA = 0,no interrupt will be acknowledged.
If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

EADC_SPI : Interrupt controller of Serial Peripheral Interface (SPI) and A/D Converter (ADC).
If EADC_SPI = 0, Disable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).
If EADC_SPI = 1, Enable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).

9. Interrupt register related with Power down wake-up: WAKE_CLKO (Non bit-Addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WAKE_CLKO	8FH	name	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CKLO	T0CKLO

PCAWAKEUP: When set and the associated-PCA interrupt control registers is configured correctly, the CEXn pin of PCA function is enabled to wake up MCU from power-down state.

RXD_PIN_IE: When set and the associated-UART interrupt control registers is configured correctly, the RXD pin (P3.0) is enabled to wake up MCU from power-down state.

T1_PIN_IE : When set and the associated-Timer1 interrupt control registers is configured correctly, the T1 pin (P3.5) is enabled to wake up MCU from power-down state.

T0_PIN_IE : When set and the associated-Timer0 interrupt control registers is configured correctly, the T1 pin (P3.4) is enabled to wake up MCU from power-down state.

T1CKLO : When set, P3.5 is enabled to be the clock output of Timer 1. The clock rate is Timer 1overflow rate divided by 2.

T0CKLO : When set, P3.4 is enabled to be the clock output of Timer 0. The clock rate is Timer 0overflow rate divided by 2.

6.3 Interrupt Priorities

Each interrupt source can also be individually programmed to one of four priority levels by setting or clearing the bits in Special Function Registers IP and IPH. A low-priority interrupt can itself be interrupted by a high-pority interrupt, but not by another low-priority interrupt. A high-priority interrupt can't be interrupted by any other interrupt source.

If two requests of different priority levels are received simultaneously, the request of higher priority level is serviced. If requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced. Thus within each priority level there is a second priority structure determined by the polling sequence,as follows:

	Source	Priority Within Level
0.	$\overline{\text{INT0}}$	(highest)
1.	Timer 0	
2.	$\overline{\text{INT1}}$	
3.	Timer 1	
4.	UART1	
5.	ADC_SPI	
6.	PCA_LVD	(lowest)

Note that the "priority within level" structure is only used to resolve *simultaneous requests of the same prionty level.*

In C language program. the interrupt polling sequence number is equal to interrupt number, for example,

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

6.4 How Interrupts Are Handled

External interrupt pins and other interrupt sources are sampled at the rising edge of each instruction *OPcode fetch cycle.* The samples are polled during the next instruction *OPcode fetch cycle*. If one of the flags was in a set condition of the first cycle, the second cycle of polling cycles will find it and the interrupt system will generate an hardware LCALL to the appropriate service routine as long as it is not blocked by any of the following conditions.

Block conditions :

- An interrupt of equal or higher priority level is already in progress.
- The current cycle(polling cycle) is not the final cycle in the execution of the instruction in progress.
- The instruction in progress is RETI or any write to the IE, IP and IPH registers.
- The ISP/IAP activity is in progress.

Any of these four conditions will block the generation of the hardware LCALL to the interrupt service routine. Condition 2 ensures that the instruction in progress will be completed before vectoring into any service routine. Condition 3 ensures that if the instruction in progress is RETI or any access to IE, IP or IPH, then at least one or more instruction will be executed before any interrupt is vectored to.

The polling cycle is repeated with the last clock cycle of each instruction cycle. Note that if an interrupt flag is active but not being responded to for one of the above conditions, if the flag is not still active when the blocking condition is removed, the denied interrupt will not be serviced. In other words, the fact that the interrupt flag was once active but not being responded to for one of the above conditions, if the flag is not still active when the blocking condition is removed, the denied interrupt will not be serviced. The interrupt flag was once active but not serviced is not kept in memory. Every polling cycle is new.

Note that if an interrupt of higher priority level goes active prior to the rising edge of the third machine cycle, then in accordance with the above rules it will be vectored to during fifth and sixth machine cycle, without any instruction of the lower priority routine having been executed.

Thus the processor acknowledges an interrupt request by executing a hardware-generated LCALL to the appropriate servicing routine. In some cases it also clears the flag that generated the interrupt, and in other cases it doesn't. It never clears the Serial Port flags. This has to be done in the user's software. It clears an external interrupt flag (IE0 or IE1) only if it was transition-activated. The hardware-generated LCALL pushes the contents of the Program Counter onto the stack (but it does not save the PSW) and reloads the PC with an address that depends on the source of the interrupt being vectored to, as shown be low.

Source	Vector Address
External Interrupt 0	0003H
Timer 0	000BH
External Interrupt 1	0013H
Timer 1	001BH
UART	0023H
ADC/SPI	002BH
PCA/LVD	0033H

Execution proceeds from that location until the RETI instruction is encountered. The RETI instruction informs the processor that this interrupt routine is no longer in progress, then pops the top two bytes from the stack and reloads the Program Counter. Execution of the interrupted program continues from where it left off.

Note that a simple RET instruction would also have returned execution to the interrupted program, but it would have left the interrupt control system thinking an interrupt was still in progress.

6.5 External Interrupts

The external sources can be programmed to be level-activated or transition-activated by clearing or setting bit IT1 or IT0 in Register TCON. If ITx = 0, external interrupt x is triggered by a detected low at the $\overline{\text{INTx}}$ pin. If ITx=1, external interrupt x is edge-triggered. In this mode if successive samples of the $\overline{\text{INTx}}$ pin show a high in one cycle and a low in the next cycle, interrupt request flag IEx in TCON is set. Flag bit IEx then requests the interrupt.

Since the external interrupt pins are sampled once each machine cycle, an input high or low should hold for at least 12 system clocks to ensure sampling. If the external interrupt is transition-activated, the external source has to hold the request pin high for at least one machine cycle, and then hold it low for at least one machine cycle to ensure that the transition is seen so that interrupt request flag IEx will be set. IEx will be automatically cleared by the CPU when the service routine is called.

If the external interrupt is level-activated, the external source has to hold the request active until the requested interrupt is actually generated. Then it has to deactivate the request before the interrupt service routine is completed, or else another interrupt will be generated.

***Example*:** Design an intrusion warning system using interrupts that sounds a 400Hz tone for 1 second (using a loudspeaker connected to P1.7) whenever a door sensor connected to $\overline{INT0}$ makes a high-to-low transition.

Assembly Language Solution

```
        ORG     0
        LJMP    MAIN                    ;3-byte instruction
        LJMP    INT0INT                 ;EXT 0 vector address
        ORG     000BH                   ;Timer 0 vector
        LJMP    T0INT
        ORG     001BH                   ;Timer 1 vector
        LJMP    T1INT
        ORG     0030H
MAIN:
        SETB    IT0                     ;negative edge activated
        MOV     TMOD,   #11H            ;16-bit timer mode
        MOV     IE,     #81H            ;enable EXT 0 only
        SJMP    $                       ;now relax
;
INT0INT:
        MOV     R7,     #20             ;20 ' 5000 us = 1 second
        SETB    TF0                     ;force timer 0 interrupt
        SETB    TF1                     ;force timer 1 interrupt
        SETB    ET0                     ;begin tone for 1 second
        SETB    ET1                     ;enable timer interrupts
        RETI
;
T0INT:
        CLR     TR0                     ;stop timer
        DJNZ    R7,     SKIP            ;if not 20th time, exit
        CLR     ET0                     ;if 20th, disable tone
        CLR     ET1                     ;disable itself
        LJMP    EXIT
SKIP:
        MOV     TH0,    #HIGH (-50000)  ;0.05sec.  delay
        MOV     TL0,     #LOW (-5000)
        SETB    TR0
EXIT:
        RETI
;
T1INT:
        CLR     TR1
        MOV     TH1,    #HIGH (-1250)   ;count for 400Hz
        MOV     TL1,    #LOW (-1250)
        CPL     P1.7                    ;music maestro!
        SETB    TR1
        RETI
        END
```

C Language Solution

```
#include <REG51.H>                      /* SFR declarations */
sbit        outbit = P1^7;              /* use variable outbit to refer to P1.7 */
unsigned char       R7;                 /* use 8-bit variable to represent R7 */
main( )
{
        IT0 = 1;                        /* negative edge activated */
        TMOD = 0x11;                    /* 16-bit timer mode */
        IE = 0x81;                      /* enable EXT 0 only */
        while(1);
}
void INT0INT(void)              interrupt 0
{
        R7 = 20;                        /* 20 x 5000us = 1 second */
        TF0 = 1;                        /* force timer 0 interrupt */
        TF1 = 1;                        /* force timer 1 interrupt */
        ET0 = 1;                        /* begin tone for 1 second */
        ET1 = 1;                        /* enable timer 1 interrupts */
                                        /* timer interrupts will do the work */
}
void T0INT(void)    interrupt 1
{
        TR0 = 0;                        /* stop timer */
        R7 = R7-1;                      /* decrement R7 */
        if (R7 == 0)                    /* if 20th time, */
        {
                ET0 = 0;                /* disable itself */
                ET1 = 0;
        }
        else
        {
                TH0 = 0x3C;             /* 0.05 sec.  delay */
                TL0 = 0xB0;
        }
}
void  T1INT (void)    interrupt 3
{
        TR0 = 0;
        TH1 = 0xFB;                     /* count for 400Hz */
        TL1 = 0x1E;
        outbit = !outbit;                /* music maestro! */
        TR1 = 1;
}
```

In the above assembly language solution, five distinct sections are the interrupt vector loactions, the main program, and the three interrupt service routines. All vector loacations contain LJMP instructions to the respective routines. The main program, starting at code address 0030H, contains only four instructions. SETB IT0 configures the door sensing interrupt input as negative-edge triggered. MOV TMOD, #11H configures both timers for mode 1, 16-bit timer mode. Only the external 0 interrupt is enabled initially (MOV IE,#81H), so a "door-open" condition is needed before any interrupt is accepted. Finally, SJMP $ puts the main program in a do-nothing loop.

When a door-open condition is sensed (by a high-to-low transition of INT0), an external 0 interrupt is generated, INT0INT begins by putting the constant 20 in R7, then sets the overflow flags for both timers to force timer interrupts to occur.

Timer interrupt will only occur, however, if the respective bits are enabled in the IE register. The next two instructions (SETB ET0 and SETB ET1) enable timer interrupts. Finally, INT0INT terminates with a RETI to the main program.

Timer 0 creates the 1 second timeout, and Timer 1 creates the 400Hz tone. After INT0INT returns to the main program, timer interrupt are immediately generated (and accepted after one excution of SJMP $). Because of the fixed polling sequence, the Timer 0 interrupt is serviced first. A 1 second timeout is created by programming 20 repetitions of a 50,000 us timeout. R7 serves as the counter. Nineteen times out of 20, T0INT operates as follows. First, Timer 0 is turned off and R7 is decremented. Then, TH0/TL is reload with -50,000, the timer is turned back on, and the interrupt is terminated. On the 20th Timer 0 interrupt, R7 is decremented to 0 (1 second has elapsed). Both timer interrupts are disabled(CLR ET0, CLR ET1)and the interrupt is terminated. No further timer interrupts will be generated until the next "door-open" condition is sensed.

The 400Hz tone is programmed using Timer 1 interrupts, 400Hz requires a period of 1/400 = 2,500 us or 1,250 high-time and 1,250 us low-time. Each timer 1 ISR simply puts -1250 in TH1/TL1, complements the port bit driving the loudspeaker, then terminates.

6.6 Response Time

The $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ levels are inverted and latched into the interrupt flags IE0 and IE1 at rising edge of every syetem clock cycle.

The Timer 0 and Timer 1 flags, TF0 and TF1, are set after which the timers overflow. The values are then polled by the circuitry at rising edge of the next system clock cycle.

If a request is active and conditions are right for it to be acknowledged, a hardware subroutine call to the requested service routine will be the next instruction to be executed. The call itself takes six system clock cycles. Thus, a minimum of seven complete system clock cycles elapse between activation of an external interrupt request and the beginning of execution of the first instruction of the service routine.

A longer response time would result if the request is blocked by one of the four previously listed conditions. If an interrupt of equal or higher priority level is already in progress, the additional wait time obviously depends on the nature of the other interrupt's service routine. If the instruction in progress is not in its final cycle, the additional wait time cannot be more than 3 cycles, since the longest instructions (LCALL) are only 6 cycles long, and if the instruction in progress is RETI or an access to IE or IP, the additional wait time cannot be more than 5 cycles (a maximum of one more cycle to complete the instruction in progress, plus 6 cycles to complete the next instruction if the instruction is LCALL).

Thus, in a single-interrupt system, the response time is always more than 7 cycles and less than 12 cycles.

6.7 Demo Programs about Interrupts (C and ASM)

6.7.1 External Interrupt 0 ($\overline{INT0}$) Demo Programs (C and ASM)

1. Demostrate External Interrupt 0 triggered by Falling Edge

C program

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------*/
/* --- STC 1T Series MCU Ext0 (Falling edge) Demo ----------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

#include "reg51.h"

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

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

        while (1);
}
```

Assembly program

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------*/
;/* --- STC 1T Series MCU Ext0(Falling edge) Demo ------------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

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

        ORG     0000H
        LJMP    MAIN

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

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

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

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

EXINT0:
        RETI

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

        END
```

2. Demostrate the Power-Down Mode waked up by Falling Edge of External Interrupt 0

C program

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by INT0 Demo -------*/
/* If you want to use the program or the program referenced in the  ------*/
/* article, please specify in which data and procedures from STC    -------*/
/*---------------------------------------------------------------------------------*/

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

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

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

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

Assembly program

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by INT0 Demo -------*/
/* If you want to use the program or the program referenced in the  ------*/
/* article, please specify in which data and procedures from STC    -------*/
/*---------------------------------------------------------------------------------*/

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

        ORG     0000H
        LJMP    MAIN

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

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

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

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

EXINT0:
        RETI

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

        END
```

6.7.2 External Interrupt 1 ($\overline{\text{INT1}}$) Demo Programs (C and ASM)

1. Demostrate External Interrupt 1 triggered by Falling Edge

C program

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series MCU Ext1(Falling edge) Demo ----------------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

#include "reg51.h"

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

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

        while (1);
}
```

Assembly program

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------*/
;/* --- STC 1T Series MCU Ext1(Falling edge) Demo ------------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

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

        ORG     0000H
        LJMP    MAIN

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

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

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

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

EXINT1:
        RETI

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

        END
```

2. Demostrate the Power-Down Mode waked up by Falling Edge of External Interrupt 1

C program

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by INT1 Demo -------*/
/* If you want to use the program or the program referenced in the  ------*/
/* article, please specify in which data and procedures from STC    ------*/
/*---------------------------------------------------------------------------------*/

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

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

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

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

Assembly program

```
/*---------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by INT1 Demo -------*/
/* If you want to use the program or the program referenced in the  ------*/
/* article, please specify in which data and procedures from STC    -------*/
/*---------------------------------------------------------------------------------*/

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

        ORG     0000H
        LJMP    MAIN

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

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

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

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

EXINT1:
        RETI

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

6.7.3 Programs of P3.4/T0 Interrupt(falling edge) used to wake up PD mode

1. C program

```
/*---------------------------------------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by T0 Demo ----------------------------------------------------------------*/
/* ---This Interrupt will borrow Timer 0 interrupt request bit TF0 and Timer 0 interrupt vector address ----*/
/* ---So Timer 0 function should be disabled when this Interrupt is enabled ---------------------------------------*/
/* ---The enable bit of this Interrupt is T0_PIN_IE / WAKE_CLKO.4 in WAKE_CLKO register ----------*/
/* If you want to use the program or the program referenced in the  ----------------------------------------------*/
/* article, please specify in which data and procedures from STC    ----------------------------------------------*/
/*---------------------------------------------------------------------------------------------------------------------*/

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

sfr WAKE_CLKO = 0x8f;

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

void main()
{
        WAKE_CLKO = 0x10;                   //enable P3.4/T0/INT falling edge wakeup MCU
                                            //from power-down mode
                                            //T0_PIN_IE (WAKE_CLKO.4) = 1
        //ET0 = 1;                          //enable T0 interrupt
        EA = 1;                             //open global interrupt switch

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

2. Assembly program

```
/* --- STC MCU International Limited ----------------------------------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by T0 Demo -----------------------------------------------*/
/* ---This Interrupt will borrow Timer 0 interrupt request bit TF0 and Timer 0 interrupt vector address ----*/
/* ---So Timer 0 function should be disabled when this Interrupt is enabled ---------------------------*/
/* ---The enable bit of this Interrupt is T0_PIN_IE / WAKE_CLKO.4 in WAKE_CLKO register ----------*/
/* If you want to use the program or the program referenced in the  ------------------------------------*/
/* article, please specify in which data and procedures from STC    ------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------*/
WAKE_CLKO   EQU 8FH
;-----------------------------------------
;interrupt vector table

            ORG     0000H
            LJMP    MAIN

            ORG     000BH                   ;T0 interrupt, interrupt 1 (location at 000BH)
            LJMP    T0INT
;-----------------------------------------
            ORG     0100H
MAIN:
            MOV     SP,#7FH                 ;initial SP
            MOV     WAKE_CLKO,    #10H      ;enable P3.4/T0/INT falling edge wakeup MCU
                                            ;from power-down mode
                                            ;T0_PIN_IE (WAKE_CLKO.4) = 1
            ;SETB   ET0                     ;enable T0 interrupt
            SETB    EA                      ;open global interrupt switch
LOOP:
            SETB    T0                      ;ready read T0 port
            JNB     T0        ,$            ;check T0
            NOP
            NOP
            MOV    PCON,     #02H           ;MCU power down
            NOP
            NOP
            CPL     P1.0
            SJMP    LOOP
;-----------------------------------------
;T0 interrupt service routine

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

6.7.4 Programs of P3.5/T1 Interrupt(falling edge) used to wake up PD mode

1. C program

```
/*---------------------------------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by T1 Demo ----------------------------------------------------------*/
/* ---This Interrupt will borrow Timer 1 interrupt request bit TF1 and Timer 1 interrupt vector address ----*/
/* ---So Timer 1 function should be disabled when this Interrupt is enabled ---------------------------------*/
/* ---The enable bit of this Interrupt is T1_PIN_IE / WAKE_CLKO.5 in WAKE_CLKO register ----------*/
/* If you want to use the program or the program referenced in the  ----------------------------------------*/
/* article, please specify in which data and procedures from STC    ----------------------------------------*/
/*---------------------------------------------------------------------------------------------------------------*/

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

sfr         WAKE_CLKO = 0x8f;

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

void main()
{
        WAKE_CLKO = 0x20;                   //enable P3.5/T1/INT falling edge wakeup MCU
                                            //from power-down mode
                                            //T1_PIN_IE / WAKE_CLKO.5 = 1
        //ET1 = 1;                          //enable T1 interrupt
        EA = 1;                             //open global interrupt switch

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

2. Assembly program

```
/* --- STC MCU International Limited ----------------------------------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by T1 Demo -----------------------------------------------*/
/* ---This Interrupt will borrow Timer 1 interrupt request bit TF1 and Timer 1 interrupt vector address ----*/
/* ---So Timer 1 function should be disabled when this Interrupt is enabled ---------------------------*/
/* ---The enable bit of this Interrupt is T1_PIN_IE / WAKE_CLKO.5 in WAKE_CLKO register ----------*/
/* If you want to use the program or the program referenced in the  ------------------------------------*/
/* article, please specify in which data and procedures from STC    ------------------------------------*/
/*----------------------------------------------------------------------------------------------------------------------*/
WAKE_CLKO   EQU 8FH

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

            ORG     0000H
            LJMP    MAIN

            ORG     001BH                   ;T1 interrupt, interrupt 3 (location at 001BH)
            LJMP    T1INT
;-----------------------------------------
            ORG     0100H
MAIN:
            MOV     SP,         #7FH        ;initial SP
            MOV     WAKE_CLKO,  #20H        ;enable P3.5/T1/INT  falling edge wakeup MCU
                                            ;from power-down mode
                                            ;T1_PIN_IE / WAKE_CLKO.5 = 1
            ;SETB   ET1                     ;enable T1 interrupt
            SETB    EA                      ;open global interrupt switch
LOOP:
            SETB    T1                      ;ready read T1 port
            JNB     T1,         $           ;check T1
            NOP
            NOP
            MOV     PCON,       #02H        ;MCU power down
            NOP
            NOP
            CPL     P1.0
            SJMP    LOOP
;-----------------------------------------
;T1 interrupt service routine

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

6.7.5 Program of P3.0/RxD Interrupt(falling edge) used to wake up PD mode

1. C program

```
/*---------------------------------------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by RxD Demo ---------------------------------------------------------*/
/* ---This Interrupt will borrow RxD interrupt request bit RI and its interrupt vector address ----------------*/
/* ---So UART function should be disabled when this Interrupt is enabled ------------------------------------*/
/* ---The enable bit of this Interrupt is RXD_PIN_IE / WAKE_CLKO.6 in WAKE_CLKO register --------*/
/* If you want to use the program or the program referenced in the  ----------------------------------------*/
/* article, please specify in which data and procedures from STC    ----------------------------------------*/
/*---------------------------------------------------------------------------------------------------------------------*/

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

typedef unsigned char BYTE;
typedef unsigned int WORD;

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

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

void main()
{
        WAKE_CLKO = 0x40;       //enable P3.0/RxD/INT falling edge wakeup MCU
                                //from power-down mode
                                //RxD_PIN_IE (WAKE_CLKO.6) = 1
        ES = 1;
        EA = 1;
```

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

2. Assembly program

```
/*---------------------------------------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by RxD Demo ---------------------------------------------------------*/
/* ---This Interrupt will borrow RxD interrupt request bit RI and its interrupt vector address ----------------*/
/* ---So UART function should be disabled when this Interrupt is enabled ----------------------------------*/
/* ---The enable bit of this Interrupt is RXD_PIN_IE / WAKE_CLKO.6 in WAKE_CLKO register --------*/
/* If you want to use the program or the program referenced in the  ----------------------------------------*/
/* article, please specify in which data and procedures from STC    ----------------------------------------*/
/*---------------------------------------------------------------------------------------------------------------------*/

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

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

        ORG     0023H
UART_ISR:
        JBC     RI,       EXIT                  ;clear RI flag
EXIT:
        RETI
;-----------------------------------------
```

```
        ORG     0100H
MAIN:
        MOV     WAKE_CLKO,      #40H        ;enable P3.0/RxD falling edge wakeup MCU
                                            ;from power-down mode
                                            ;RxD_PIN_IE (WAKE_CLKO.6) = 1
        SETB    ES
        SETB    EA

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

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

        END
```

6.7.6 Program of PCA Interrupt used to wake up Power Down mode

1. C program

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by PCA Demo ---------------------------------*/
/* If you want to use the program or the program referenced in the  ----------*/
/* article, please specify in which data and procedures from STC    ----------*/
/*---------------------------------------------------------------------------------------*/

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

typedef unsigned char BYTE;
typedef unsigned int WORD;

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

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

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

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

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

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

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

2 **Assembly program**

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------*/
/* --- STC 1T Series MCU Power-Down wakeup by PCA Demo ---------------------------------*/
/* If you want to use the program or the program referenced in the  ----------*/
/* article, please specify in which data and procedures from STC    ----------*/
/*---------------------------------------------------------------------------------------*/

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

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

PCA_LED         BIT     P1.1            ;PCA test LED
CCP0            BIT     P1.3
```

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

        ORG     003BH
PCA_ISR:
        CLR     CCF0                    ;Clear interrupt flag
        CPL     PCA_LED                 ;toggle the test pin while CCP0(P3.7) have a falling edge
        RETI
;---------------------------------------
        ORG     0100H
MAIN:
        MOV     CCON,   #0              ;Initial PCA control register
                                        ;PCA timer stop running
                                        ;Clear CF flag
                                        ;Clear all module interrupt flag
        CLR     A                       ;
        MOV     CL,     A               ;Reset PCA base timer
        MOV     CH,     A               ;
        MOV     CMOD,   #00H            ;Set PCA timer clock source as Fosc/12
                                        ;Disable PCA timer overflow interrupt
        MOV     CCAPM0,         #11H    ;PCA module-0 capture by a falling edge on CCP0(P1.3)
                                        ;and enable PCA interrupt
;       MOV     CCAPM0,         #21H    ;PCA module-0 capture by a rising edge on CCP0(P1.3)
                                        ;and enable PCA interrupt
;       MOV     CCAPM0,         #31H    ;PCA module-0 capture by a transition (falling/rising edge)
                                        ;on CCP0(P1.3) and enable PCA interrupt
;-------------------------------
        MOV     WAKE_CLKO,      #80H    ;enable PCA falling/raising edge wakeup MCU from
                                        ;power-down mode
        SETB    CR                      ;PCA timer start run
        SETB    EA

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

Chapter 7. Timer/Counter 0/1

Timer 0 and timer 1 are like the ones in the conventional 8051, both of them can be individually configured as timers or event counters.

In the “Timer” function, the register is incremented every 12 system clocks or every system clock depending on AUXR.7(T0x12) bit and AUXR.6(T1x12). In the default state, it is fully the same as the conventional 8051. In the x12 mode, the count rate equals to the system clock.

In the “Counter” function, the register is incremented in response to a 1-to-0 transition at its corresponding external input pin, T0 or T1. In this function, the external input is sampled once at the positive edge of every clock cycle. When the samples show a high in one cycle and a low in the next cycle, the count is incremented. The new count value appears in the register during at the end of the cycle following the one in which the transition was detected. Since it takes 2 machine cycles (24 system clocks) to recognize a l-to-0 transition, the maximum count rate is 1/24 of the system clock. There are no restrictions on the duty cycle of the external input signal, but to ensure that a given level is sampled at least once before it changes, it should be held for at least one full machine cycle.

In addition to the “Timer” or “Counter” selection, Timer 0 and Timer 1 have four operating modes from which to select. The “Timer” or “Counter” function is selected by control bits C/$\overline{T}$ in the Special Function Register TMOD. These two Timer/Counter have four operating modes, which are selected by bit-pairs (M1, M0) in TMOD. Modes 0, 1, and 2 are the same for both Timer/Counters. Mode 3 is different.The four operating modes are described in the following text.

7.1 Special Function Registers about Timer/Counter

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
TCON	Timer Control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
TMOD	Timer Mode	89H	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	0000 0000B
TL0	Timer Low 0	8AH									0000 0000B
TL1	Timer Low 1	8BH									0000 0000B
TH0	Timer High 0	8CH									0000 0000B
TH1	Timer High 1	8DH									0000 0000B
AUXR	Auxiliary register	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
WAKE_CLKO	CLK_Output Power down Wake-up control register	8FH	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO	0000 xx00B

1. TCON register: Timer/Counter Control Register (Bit-Addressable)

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

TF1: Timer/Counter 1 Overflow Flag. Set by hardware on Timer/Counter 1 overflow. The flag can be cleared by software but is automatically cleared by hardware when processor vectors to the Timer 1 interrupt routine.
If TF1 = 0, No Timer 1 overflow detected.
If TF1 = 1, Timer 1 has overflowed.

TR1: Timer/Counter 1 Run Control bit. Set/cleared by software to turn Timer/Counter on/off.
If TR1 = 0, Timer 1 disabled.
If TR1 = 1, Timer 1 enabled.

TF0: Timer/Counter 0 Overflow Flag. Set by hardware on Timer/Counter 0 overflow. The flag can be cleared by software but is automatically cleared by hardware when processor vectors to the Timer 0 interrupt routine.
If TF0 = 0, No Timer 0 overflow detected.
If TF0 = 1, Timer 0 has overflowed.

TR0: Timer/Counter 0 Run Control bit. Set/cleared by software to turn Timer/Counter on/off.
If TR0 = 0, Timer 0 disabled.
If TR0 = 1, Timer 0 enabled.

IE1: External Interrupt 1 Edge flag. Set by hardware when external interrupt edge/level defined by IT1 is detected. The flag can be cleared by software but is automatically cleared when the external interrupt 1 service routine has been processed.

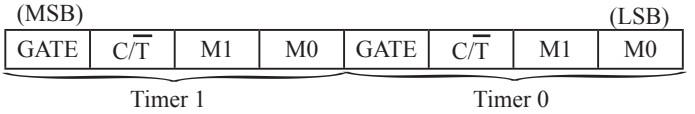
IT1: External Intenupt 1 Type Select bit. Set/cleared by software to specify falling edge/low level triggered external interrupt 1.
If IT1 = 0, $\overline{\text{INT1}}$ is low level triggered.
If IT1 = 1, $\overline{\text{INT1}}$ is edge triggered.

IE0: External Interrupt 0 Edge flag. Set by hardware when external interrupt edge/level defined by IT0 is detected. The flag can be cleared by software but is automatically cleared when the external interrupt 0 service routine has been processed.

IT0: External Intenupt 0 Type Select bit. Set/cleared by software to specify falling edge/low level triggered external interrupt 0.
If IT0 = 0, $\overline{\text{INT0}}$ is low level triggered.
If IT0 = 1, $\overline{\text{INT0}}$ is edge triggered.

2. TMOD register: Timer/Counter Mode Register

TMOD address: 89H (Non bit-addressable)



GATR/TMOD.7: Timer/Counter Gate Control.

If GATE/TMOD.7=0,Timer/Counter 1 enabled when TR1 is set irrespective of $\overline{INT1}$ logic level;

If GATE/TMOD.7=1, Timer/Counter 1 enabled only when TR1 is set AND $\overline{INT1}$ pin is high.

C/$\overline{T}$/TMOD.6: Timer/Counter 1 Select bit.

If C/$\overline{T}$/TMOD.6=0,Timer/Counter 1 is set for Timer operation (input from internal system clock);

If C/$\overline{T}$/TMOD.6=0,Timer/Counter 1 is set for Counter operation (input from external T1 pin).

M1/TMOD.5 ~ M0/TMOD.4: Timer 1 Mode Select bits.

M1	M0	Operating Mode
0	0	Mode 0: 13-bit Timer/Counter for Timer 1
0	1	Mode 1: 16-bit Timer/Counter. TH1and TL1 are cascaded; there is no prescaler.
1	0	Mode 2: 8-bit auto-reload Timer/Counter. TH1 holds a value which is to be reloaded into TL1 each time it overflows.
1	1	Timer/Counter 1 stopped

GATR/TMOD.3: Timer/Counter Gate Control.

If GATE/TMOD.3=0,Timer/Counter 0 enabled when TR0 is set irrespective of $\overline{INT0}$ logic level;

If GATE/TMOD.3=1, Timer/Counter 0 enabled only when TR0 is set AND $\overline{INT0}$ pin is high.

C/$\overline{T}$/TMOD.2: Timer/Counter 0 Select bit.

If C/$\overline{T}$/TMOD.2=0,Timer/Counter 0 is set for Timer operation (input from internal system clock);

If C/$\overline{T}$/TMOD.2=0,Timer/Counter 0 is set for Counter operation (input from external T0 pin).

M1/TMOD.1 ~ M0/TMOD.0: Timer 0 Mode Select bits.

M1	M0	Operating Mode
0	0	Mode 0: 13-bit Timer/Counter for Timer 0
0	1	Mode 1: 16-bit Timer/Counter. TH0 and TL0 are cascaded; there is no prescaler.
1	0	Mode 2: 8-bit auto-reload Timer/Counter. TH0 holds a value which is to be reloaded into TL0 each time it overflows.
1	1	Mode3: TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits TH0 is an 8-bit timer only controlled by Timer 1 control bits.

3. AUXR: Auxiliary register (Non bit-addressable)

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

T0x12 : Timer 0 clock source bit.
0 : The clock source of Timer 0 is SYSclk/12. It will compatible to the traditional 80C51 MCU
1 : The clock source of Timer 0 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

T1x12 : Timer 1 clock source bit.
0 : The clock source of Timer 1 is SYSclk/12. It will compatible to the traditional 80C51 MCU
1 : The clock source of Timer 1 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

UART_M0x6 : Baud rate select bit of UART1 while it is working under Mode-0
0 : The baud-rate of UART in mode 0 is SYSclk/12.
1 : The baud-rate of UART in mode 0 is SYSclk/2.

EADCI : Enable/Disable interrupt from A/D converter
0 : (default) Inhibit the ADC functional block to generate interrupt to the MCU
1 : Enable the ADC functional block to generate interrupt to the MCU

ESPI : Enable/Disable interrupt from Serial Peripheral Interface (SPI)
0 : (default)Inhibit the SPI functional block to generate interrupt to the MCU
1 : Enable the SPI functional block to generate interrupt to the MCU

ELVDI : Enable/Disable interrupt from low-voltage sensor
0 : (default)Inhibit the low-voltage sensor functional block to generate interrupt to the MCU
1 : Enable the low-voltage sensor functional block to generate interrupt to the MCU

4. WAKE_CLKO: CLK_Output Power down Wake-up control register (Non bit-Addressable)

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

PCAWAKEUP: When set and the associated-PCA interrupt control registers is configured correctly, the CEXn pin of PCA function is enabled to wake up MCU from power-down state.

RXD_PIN_IE: When set and the associated-UART interrupt control registers is configured correctly, the RXD pin (P3.0) is enabled to wake up MCU from power-down state.

T1_PIN_IE : When set and the associated-Timer1 interrupt control registers is configured correctly, the T1 pin (P3.5) is enabled to wake up MCU from power-down state.

T0_PIN_IE : When set and the associated-Timer0 interrupt control registers is configured correctly, the T1 pin (P3.4) is enabled to wake up MCU from power-down state.

T1CKLO : When set, P3.5 is enabled to be the clock output of Timer 1. The clock rate is Timer 1overflow rate divided by 2.

T0CKLO : When set, P3.4 is enabled to be the clock output of Timer 0. The clock rate is Timer 0overflow rate divided by 2.

7.2 Timer/Counter 0 Mode of Operation (Compatible with traditional 8051 MCU)

Timer/Counter 0 can be configured for four modes by setting M1(TMOD.1) and M0(TMOD.0) in sepcial function register TMOD.

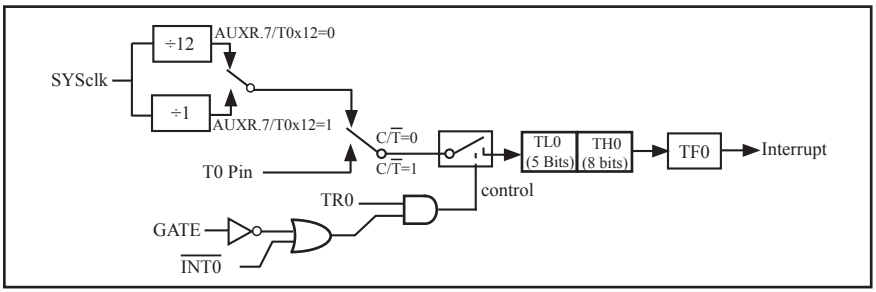
7.2.1 Mode 0 (13-bit Timer/Counter)

Mode 0

In this mode, the timer 0 is configured as a 13-bit timer/counter. As the count rolls over from all 1s to all 0s, it sets the timer interrupt flag TF0. The counted input is enabled to the timer when TR0 = 1 and either GATE=0 or $\overline{INT0}$ = 1.(Setting GATE = 1 allows the Timer to be controlled by external input $\overline{INT0}$, to facilitate pulse width measurements.) TR0 is a control bit in the Special Function Register TCON. GATE is in TMOD.

The 13-Bit register consists of all 8 bits of TH0 and the lower 5 bits of TL0. The upper 3 bits of TL0 are indeterminate and should be ignored. Setting the run flag (TR0) does not clear the registers.

There are two different GATE bits. one for Timer 1 (TMOD.7) and one for Timer 0 (TMOD.3).



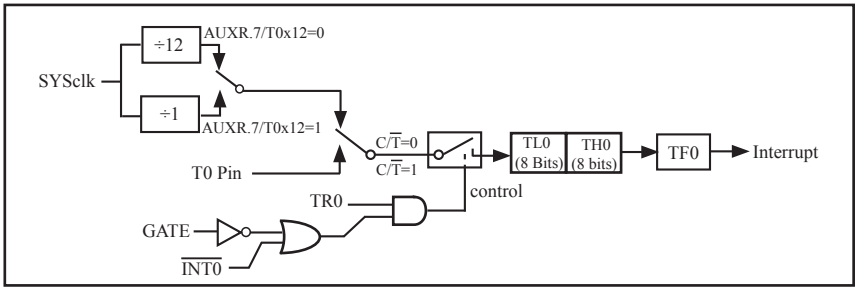
Timer/Counter 0 Mode 0: 13-Bit Timer/Counter

7.2.2 Mode 1 (16-bit Timer/Counter) and Demo Programs (C and ASM)

In this mode, the timer register is configured as a 16-bit register. As the count rolls over from all 1s to all 0s, it sets the timer interrupt flag TF0. The counted input is enabled to the timer when TR0 = 1 and either GATE=0 or $\overline{INT0}$ = 1.(Setting GATE = 1 allows the Timer to be controlled by external input $\overline{INT0}$, to facilitate pulse width measurements.) TR0 is a control bit in the Special Function Register TCON. GATE is in TMOD.

The 16-Bit register consists of all 8 bits of TH0 and the lower 8 bits of TL0. Setting the run flag (TR0) does not clear the registers.

Mode 1 is the same as Mode 0, except that the timer register is being run with all 16 bits.



Timer/Counter 0 Mode 1 : 16-Bit Timer/Counter

There are two simple programs that demostrates Timer 0 as 16-bit Timer/Counter, one written in C language while other in Assembly language.

C Program:

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series 16-bit Timer Demo -------------------------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

#include "reg51.h"

typedef unsigned char BYTE;
typedef unsigned int WORD;

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

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

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

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

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

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

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

/* main program */
void main()
{
#ifdef MODE1T
        AUXR = 0x80;                             //timer0 work in 1T mode
#endif
        TMOD = 0x01;                             //set timer0 as mode1 (16-bit)
        TL0 = T1MS;                              //initial timer0 low byte
        TH0 = T1MS >> 8;                         //initial timer0 high byte
        TR0 = 1;                                 //timer0 start running
        ET0 = 1;                                 //enable timer0 interrupt
        EA = 1;                                  //open global interrupt switch
        count = 0;                               //initial counter

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

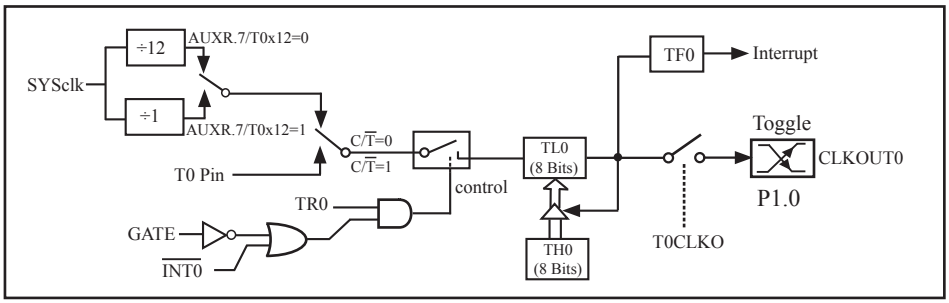
Assembly Program:

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------*/
;/* --- STC 1T Series 16-bit Timer Demo ----------------------------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

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

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

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

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

;------------------------------------------------
        ORG     0000H
        LJMP    MAIN
        ORG     000BH
        LJMP    TM0_ISR

;------------------------------------------------
;/* main program */
MAIN:
#ifdef MODE1T
        MOV     AUXR,   #80H                    ;timer0 work in 1T mode
#endif
        MOV     TMOD,   #01H                    ;set timer0 as mode1 (16-bit)
        MOV     TL0,    #LOW T1MS               ;initial timer0 low byte
        MOV     TH0,    #HIGH T1MS              ;initial timer0 high byte
        SETB    TR0                             ;timer0 start running
        SETB    ET0                             ;enable timer0 interrupt
        SETB    EA                              ;open global interrupt switch
        CLR     A
        MOV     COUNT,  A
        MOV     COUNT+1,A                       ;initial counter
        SJMP    $
```

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

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

        END
```

7.2.3 Mode 2 (8-bit Auto-Reload Mode) and Demo Programs (C and ASM)

Mode 2 configures the timer register as an 8-bit counter(TL0) with automatic reload. Overflow from TL0 not only set TF0, but also reload TL0 with the content of TH0, which is preset by software. The reload leaves TH0 unchanged.



Timer/Counter 0 Mode 2: 8-Bit Auto-Reload

STC12C2052AD is able to generate a programmable clock output on P1.0. When T0CLKO/WAKE_CLKO.0 bit in WAKE_CLKO SFR is set, T0 timer overflow pulse will toggle P1.0 latch to generate a 50% duty clock. The frequency of clock-out = T0 overflow rate/2.

If C/$\overline{T}$(TMOD.2) = 0, Timer/Counter 0 is set for Timer operation (input from internal system clock), the Frequency of clock-out is as following :

(SYSclk) / (256 – TH0) / 2, when AUXR.7 / T0x12=1

or **(SYSclk / 12) / (256 – TH0) / 2 , when AUXR.7 / T0x12=0**

If C/$\overline{T}$(TMOD.2) = 1, Timer/Counter 0 is set for Conter operation (input from external P3.4/T0 pin), the Frequency of clock-out is as following :

T0_Pin_CLK / (256-TH0) / 2

;T0 Interrupt (falling edge) Demo programs, where T0 operated in Mode 2 (8-bit auto-relaod mode)
; The Timer Interrupt can not wake up MCU from Power-Down mode in the following programs

1. C program

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

#include "reg51.h"

sfr         AUXR = 0x8e;                    //Auxiliary register

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

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

        while (1);
}
```

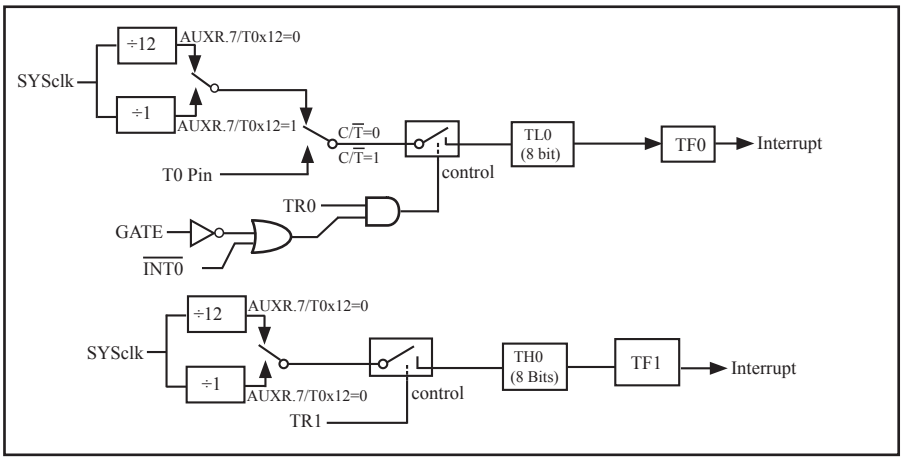
2. Assembly program

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

AUXR    DATA    08EH                    ;Auxiliary register

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

        ORG     0000H
        LJMP    MAIN

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

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

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

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

T0INT:
        RETI

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

        END
```

7.2.4 Mode 3 (Two 8-bit Timers/Couters)

Timer 1 in Mode 3 simply holds its count, the effect is the same as setting TR1 = 0. Timer 0 in Mode 3 established TL0 and TH0 as two separate 8-bit counters. TL0 use the Timer 0 control bits: C/$\overline{T}$,GATE,TR0, $\overline{INT0}$ and TF0. TH0 is locked into a timer function (counting machine cycles) and takes over the use of TR1 from Tmer 1. Thus, TH0 now controls the "Timer 1" interrupt.

Mode 3 is provided for applications requiring an extra 8-bit timer or counter. When Timer 0 is in Mode 3, Timer 1 can be turned on and off by switching it out of and into its own Mode 3, or can still be used by the serial port as a baud rate generator, or in fact, in any application not requiring an interrupt.



Timer/Counter 0 Mode 3: Two 8-Bit Timers/Counters

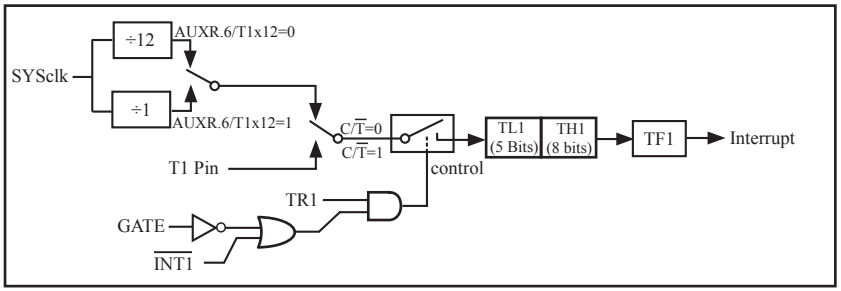
7.3 Timer/Counter 1 Mode of Operation

Timer/Counter 1 can be configured for three modes by setting M1(TMOD.5) and M0(TMOD.4) in sepcial function register TMOD.

7.3.1 Mode 0 (13-bit Timer/Counter)

In this mode, the timer register is configured as a 13-bit register. As the count rolls over from all 1s to all 0s, it sets the timer interrupt flag TF1. The counted input is enabled to the timer when TR1 = 1 and either GATE=0 or $\overline{INT1}$= 1.(Setting GATE = 1 allows the Timer to be controlled by external input $\overline{INT1}$, to facilitate pulse width measurements.) TR0 is a control bit in the Special Function Register TCON. GATE is in TMOD.

The 13-Bit register consists of all 8 bits of TH1 and the lower 5 bits of TL1. The upper 3 bits of TL1 are indeterminate and should be ignored. Setting the run flag (TR1) does not clear the registers.



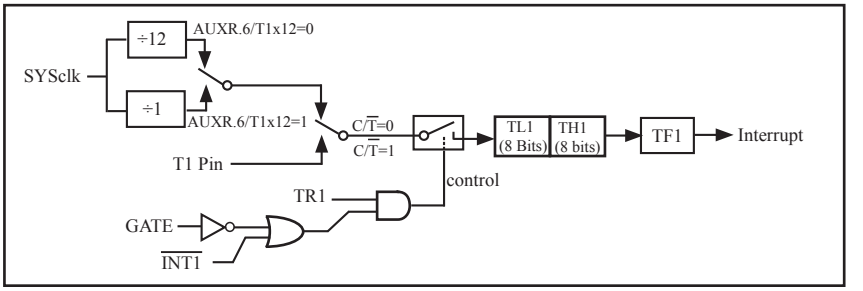
Timer/Counter 1 Mode 0: 13-Bit Timer/Counter

7.3.2 Mode 1 (16-bit Timer/Counter) and Demo Programs (C and ASM)

In this mode, the timer register is configured as a 16-bit register. As the count rolls over from all 1s to all 0s, it sets the timer interrupt flag TF1. The counted input is enabled to the timer when TR1 = 1 and either GATE=0 or $\overline{INT1}$ = 1.(Setting GATE = 1 allows the Timer to be controlled by external input $\overline{INT1}$, to facilitate pulse width measurements.) TRl is a control bit in the Special Function Register TCON. GATE is in TMOD.

The 16-Bit register consists of all 8 bits of THl and the lower 8 bits of TL1. Setting the run flag (TR1) does not clear the registers.

Mode 1 is the same as Mode 0, except that the timer register is being run with all 16 bits.



Timer/Counter 1 Mode 1 : 16-Bit Timer/Counter

There are another two simple programs that demostrates Timer 1 as 16-bit Timer/Counter, one written in C language while other in Assembly language.

1. C Program

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ------------------------------------*/
/* --- STC 1T Series 16-bit Timer Demo ----------------------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

#include "reg51.h"

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

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

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

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

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

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

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

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

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

/* main program */
void main()
{
#ifdef MODE1T
        AUXR = 0x40;                    //timer1 work in 1T mode
#endif
        TMOD = 0x10;                    //set timer1 as mode1 (16-bit)
        TL1 = T1MS;                     //initial timer1 low byte
        TH1 = T1MS >> 8;                //initial timer1 high byte
        TR1 = 1;                        //timer1 start running
        ET1 = 1;                        //enable timer1 interrupt
        EA = 1;                         //open global interrupt switch
        count = 0;                      //initial counter

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

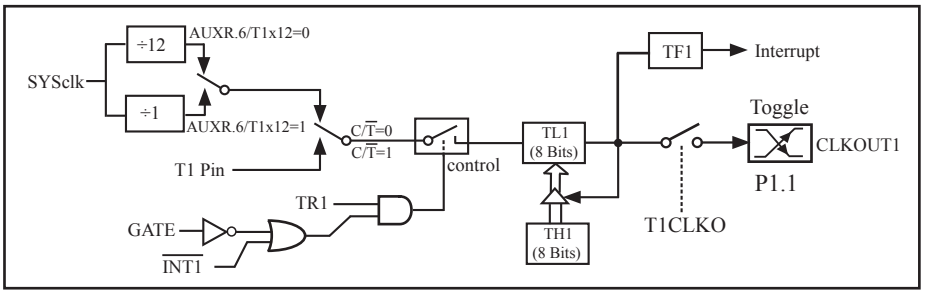
2. Assembly Program

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------*/
;/* --- STC 1T Series 16-bit Timer Demo ----------------------------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

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

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

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

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

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

            ORG     0000H
            LJMP    MAIN
            ORG     001BH
            LJMP    TM1_ISR

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

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

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

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

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

        END
```

7.3.3 Mode 2 (8-bit Auto-Reload Mode) and Demo Programs (C and ASM)

Mode 2 configures the timer register as an 8-bit counter(TL1) with automatic reload. Overflow from TL1 not only set TFx, but also reload TL1 with the content of TH1, which is preset by software. The reload leaves TH1 unchanged.



Timer/Counter 1 Mode 2: 8-Bit Auto-Reload

STC12C2052AD is able to generate a programmable clock output on P1.1. When T1CLKO/WAKE_CLKO.1 bit in WAKE_CLKO SFR is set, T1 timer overflow pulse will toggle P1.1 latch to generate a 50% duty clock. The frequency of clock-out = T1 overflow rate/2.

If C/$\overline{T}$(TMOD.6) = 0, Timer/Counter 1 is set for Timer operation (input from internal system clock), the Frequency of clock-out is as following :

(SYSclk) / (256 – TH1) / 2, when AUXR.6 / T0x12=1

or **(SYSclk / 12) / (256 – TH1) / 2 , when AUXR.6 / T0x12=0**

If C/$\overline{T}$(TMOD.6) = 1, Timer/Counter 1 is set for Conter operation (input from external P3.5/T1 pin), the Frequency of clock-out is as following :

T1_Pin_CLK / (256-TH1) / 2

;T1 Interrupt (falling edge) Demo programs, where T1 operated in Mode 2 (8-bit auto-relaod mode)
; The Timer Interrupt can not wake up MCU from Power-Down mode in the following programs

1. C program

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

#include "reg51.h"

sfr AUXR = 0x8e;                                    //Auxiliary register

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

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

        while (1);
}
```

2. Assembly program

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

AUXR    DATA    08EH                    ;Auxiliary register

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

        ORG     0000H
        LJMP    MAIN

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

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

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

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

T1INT:
        RETI

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

        END
```

7.4 Programmable Clock Output and Demo Programs (C and ASM)

STC12C2052AD series MCU have two channel programmable clock outputs, they are Timer 0 programmable clock output CLKOUT0(P1.0) and Timer 1 programmable clock output CLKOUT1(P1.1).

There are some SFRs about programmable clock output as shown below.

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
AUXR	Auxiliary register	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
WAKE_CLKO	CLK_Output Power down Wake-up control register	8FH	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO	0000 xx00B

The satement (used in C language) of Special function registers AUXR/WAKE_CLKO:

```
sfr     AUXR      =  0x8E;     //The address statement of Special function register AUXR
sfr     WAKE_CLKO =  0x8F;     //The address statement of SFR WAKE_CLKO
```

The satement (used in Assembly language) of Special function registers AUXR/WAKE_CLKO:

```
AUXR      EQU  0x8E      ;The address statement of Special function register AUXR
WAKE_CLKO EQU  0x8F      ;The address statement of SFR WAKE_CLKO
```

1. AUXR: Auxiliary register (Non bit-addressable)

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

T0x12 : Timer 0 clock source bit.
- 0 : The clock source of Timer 0 is SYSclk/12. It will compatible to the traditional 80C51 MCU
- 1 : The clock source of Timer 0 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

T1x12 : Timer 1 clock source bit.
- 0 : The clock source of Timer 1 is SYSclk/12. It will compatible to the traditional 80C51 MCU
- 1 : The clock source of Timer 1 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

UART_M0x6 : Baud rate select bit of UART1 while it is working under Mode-0
- 0 : The baud-rate of UART in mode 0 is SYSclk/12.
- 1 : The baud-rate of UART in mode 0 is SYSclk/2.

UART_M0x6 : Baud rate select bit of UART1 while it is working under Mode-0
- 0 : The baud-rate of UART in mode 0 is SYSclk/12.
- 1 : The baud-rate of UART in mode 0 is SYSclk/2.

EADCI : Enable/Disable interrupt from A/D converter
- 0 : (default) Inhibit the ADC functional block to generate interrupt to the MCU
- 1 : Enable the ADC functional block to generate interrupt to the MCU

ESPI : Enable/Disable interrupt from Serial Peripheral Interface (SPI)
- 0 : (default)Inhibit the SPI functional block to generate interrupt to the MCU
- 1 : Enable the SPI functional block to generate interrupt to the MCU

ELVDI : Enable/Disable interrupt from low-voltage sensor
- 0 : (default)Inhibit the low-voltage sensor functional block to generate interrupt to the MCU
- 1 : Enable the low-voltage sensor functional block to generate interrupt to the MCU

2. WAKE_CLKO: CLK_Output Power down Wake-up control register (Non bit-Addressable)

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

PCAWAKEUP: When set and the associated-PCA interrupt control registers is configured correctly, the CEXn pin of PCA function is enabled to wake up MCU from power-down state.

RXD_PIN_IE: When set and the associated-UART interrupt control registers is configured correctly, the RXD pin (P3.0) is enabled to wake up MCU from power-down state.

T1_PIN_IE : When set and the associated-Timer1 interrupt control registers is configured correctly, the T1 pin (P3.5) is enabled to wake up MCU from power-down state.

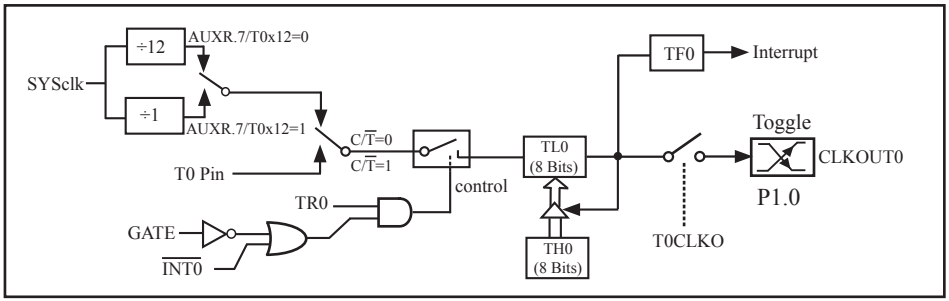
T0_PIN_IE : When set and the associated-Timer0 interrupt control registers is configured correctly, the T1 pin (P3.4) is enabled to wake up MCU from power-down state.

LVD_WAKE: When set and the associated-LVD interrupt control registers is configured correctly, the CMPIN pin is enabled to wake up MCU from power-down state.

T1CKLO : When set, P3.5 is enabled to be the clock output of Timer 1. The clock rate is Timer 1overflow rate divided by 2.

T0CKLO : When set, P3.4 is enabled to be the clock output of Timer 0. The clock rate is Timer 0overflow rate divided by 2.

7.4.1 Timer 0 Programmable Clock-out on P1.0 and Demo Program(C and ASM)



Timer/Counter 0 Mode 2: 8-Bit Auto-Reload

STC12C2052AD is able to generate a programmable clock output on P1.0. When T0CLKO/WAKE_CLKO.0 bit in WAKE_CLKO SFR is set, T0 timer overflow pulse will toggle P1.0 latch to generate a 50% duty clock. The frequency of clock-out = T0 overflow rate/2.

If C/$\overline{T}$(TMOD.2) = 0, Timer/Counter 0 is set for Timer operation (input from internal system clock), the Frequency of clock-out is as following :

(SYSclk) / (256 – TH0) / 2, when AUXR.7 / T0x12=1

or **(SYSclk / 12) / (256 – TH0) / 2 , when AUXR.7 / T0x12=0**

If C/$\overline{T}$ (TMOD.2) = 1, Timer/Counter 0 is set for Conter operation (input from external P3.4/T0 pin), the Frequency of clock-out is as following :

T0_Pin_CLK / (256-TH0) / 2

The following programs demostrate Program Clock Output on P1.0 pin when Timer 0 operates as 8-bit auto-reload Timer/Counter.

1. C Program:

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series Programmable Clock Output Demo ------------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

#include "reg51.h"
//-----------------------------------------------
/* define constants */
#define   FOSC    18432000L
//#define  MODE    1T              //Timer clock mode, comment this line is 12T mode, uncomment is 1T mode
```

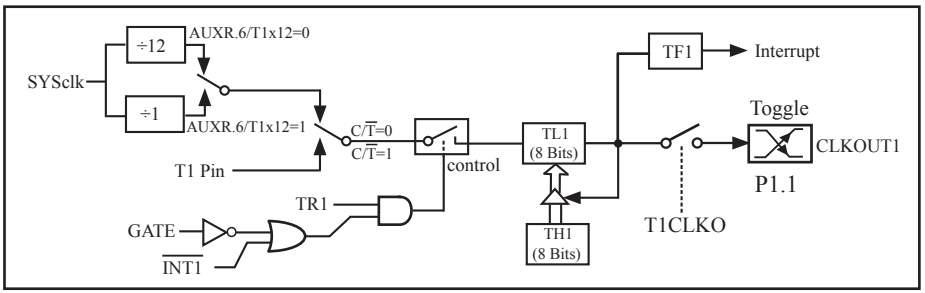
```
#ifdef     MODE    1T
#define    F38_4KHz          (256-FOSC/2/38400)          //38.4KHz frequency calculation method of 1T mode
#else
#define    F38_4KHz          (256-FOSC/2/12/38400)       //38.4KHz frequency calculation method of 12T mode
#endif

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

//-----------------------------------------------
/* main program */
void main()
{
#ifdef MODE1T
           AUXR    = 0x80;                    //timer0 work in 1T mode
#endif
           TMOD    = 0x02;                    //set timer0 as mode2 (8-bit auto-reload)
           TL0     = F38_4KHz;                //initial timer0
           TH0     = F38_4KHz;                //initial timer0
           TR0     = 1;                       //timer0 start running
           WAKE_CLKO = 0x01;                  //enable timer0 clock output

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

2. Assembly Program:

```
;/*-------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------*/
;/* --- STC 1T Series Programmable Clock Output Demo ----------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*-------------------------------------------------------------------------------*/

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

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

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

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

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

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

          SJMP    $

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

          END
```

7.4.2 Timer 1 Programmable Clock-out on P1.1 and Demo Program(C and ASM)



Timer/Counter 1 Mode 2: 8-Bit Auto-Reload

STC12C2052AD is able to generate a programmable clock output on P1.1. When T1CLKO/WAKE_CLKO.1 bit in WAKE_CLKO SFR is set, T1 timer overflow pulse will toggle P1.1 latch to generate a 50% duty clock. The frequency of clock-out = T1 overflow rate/2.

If C/$\overline{T}$(TMOD.6) = 0, Timer/Counter 1 is set for Timer operation (input from internal system clock), the Frequency of clock-out is as following :

(SYSclk) / (256 – TH1) / 2, when AUXR.6 / T0x12=1

or **(SYSclk / 12) / (256 – TH1) / 2 , when AUXR.6 / T0x12=0**

If C/$\overline{T}$(TMOD.6) = 1, Timer/Counter 1 is set for Conter operation (input from external P3.5/T1 pin), the Frequency of clock-out is as following :

T1_Pin_CLK / (256-TH1) / 2

The following programs demostrate Program Clock Output on Timer 1 pin when Timer 1 operates as 8-bit auto-reload Timer/Counter.

1. C Program:

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series Programmable Clock Output Demo ------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

#include "reg51.h"

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

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

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

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

/* main program */
void main()
{
#ifdef MODE1T
          AUXR    =         0x40;                                 //timer1 work in 1T mode
#endif
          TMOD    =         0x20;                                 //set timer1 as mode2 (8-bit auto-reload)
          TL1     =         F38_4KHz;                             //initial timer1
          TH1     =         F38_4KHz;                             //initial timer1
          TR1     =         1;                                    //timer1 start running
          WAKE_CLKO =   0x02;                                     //enable timer1 clock output

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

2. Assembly Program:

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------*/
;/* --- STC 1T Series Programmable Clock Output Demo ---------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

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

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

;/* define SFR */
AUXR            DATA    08EH            ;Auxiliary register
WAKE_CLKO       DATA    08FH            ;wakeup and clock output control register
T1CLKO          BIT     P1.1            ;timer1 clock output pin

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

        ORG     0000H
        LJMP    MAIN

;------------------------------------------------
;/* main program */
MAIN:
#ifdef MODE1T
        MOV     AUXR,   #40H                    ;timer1 work in 1T mode
#endif
        MOV     TMOD,   #20H                    ;set timer1 as mode2 (8-bit auto-reload)
        MOV     TL1,    #F38_4KHz               ;initial timer1
        MOV     TH1,    #F38_4KHz               ;initial timer1
        SETB    TR1
        MOV     WAKE_CLKO,      #02H            ;enable timer1 clock output

        SJMP    $

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

        END
```

7.5 Application note for Timer in practice

(1) Real-time Timer

Timer/Counter start running, When the Timer/Counter is overflow, the interrupt request generated, this action handle by the hardware automatically, however, the process which from propose interrupt request to respond interrupt request requires a certain amount of time, and that the delay interrupt request on-site with the environment varies, it normally takes three machine cycles of delay, which will bring real-time processing bias. In most occasions, this error can be ignored, but for some real-time processing applications, which require compensation.

Such as the interrupt response delay, for timer mode 0 and mode 1, there are two meanings: the first, because of the interrupt response time delay of real-time processing error; the second, if you require multiple consecutive timing, due to interruption response delay, resulting in the interrupt service program once again sets the count value is delayed by several count cycle.

If you choose to use Timer/Counter mode 1 to set the system clock, these reasons will produce real-time error for this situation, you should use dynamic compensation approach to reducing error in the system clock, compensation method can refer to the following example program.

```
…
CLR     EA              ;disable interrupt
MOV     A,      TLx     ;read TLx
ADD     A,      #LOW    ;LOW is low byte of compensation value
MOV     TLx,    A       ;update TLx
MOV     A,      THx     ;read THx
ADDC    A,      #HIGH   ;HIGH is high byte of compensation value
MOV     THx,    A       ;update THx
SETB    EA              ;enable interrupt
…
```

(2) Dynamic read counts

When dynamic read running timer count value, if you do not pay attention to could be wrong, this is because it is not possible at the same time read the value of the TLx and THx. For example the first reading TLx then THx, because the timer is running, after reading TLx, TLx carry on the THx produced, resulting in error; Similarly, after the first reading of THx then TLx, also have the same problems.

A kind of way avoid reading wrong is first reading THx then TLx and read THx once more, if the THx twice to read the same value, then the read value is correct, otherwise repeat the above process. Realization method reference to the following example code.

```
        …
RDTM:   MOV     A,      THx             ;save THx to ACC
        MOV     R0,     TLx             ;save TLx to R0
        CJNE    A,      THx,    RDTM    ;read THx again and compare with the previous value
        MOV     R1,     A               ;save THx to R1
        …
```

Chapter 8. UART with Enhanced Function

The serial port is full duplex,meaning it can transmit and receive simultaneously. It is also receive-buffered, meaning it can commence reception of a second byte before a previously received byte has been read from the reeeive register. (However,if the first byte still hasn't been read by the time reception of the second byte is complete, one of the bytes will be lost). The serial port receive and transmit share the same SFR – SBUF, but actually there is two SBUF in the chip, one is for transmit and the other is for receive.

The serial port(UART) can be operated in 4 different modes: Mode 0 provides synchronous communication while Modes 1, 2, and 3 provide asynchronous communication. The asynchronous communication operates as a full-duplex Universal Asynchronous Receiver and Transmitter (UART), which can transmit and receive simultaneously and at different baud rates.

Serial communiction involves the transimission of bits of data through only one communication line. The data are transimitted bit by bit in either synchronous or asynchronous format. Synchronous serial communication transmits ont whole block of characters in syschronization with a reference clock while asynchronous serial communication randomly transmits one character at any time, independent of any clock.

8.1 Special Function Registers about UART

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
AUXR	Auxiliary register	8EH	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	0000 00xxB
SCON	Serial Control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	0000 0000B
SBUF	Serial Buffer	99H									xxxx xxxxB
PCON	Power Control	87H	SMOD	SMOD0	LVDF	POF	GF1	GF0	PD	IDL	0011 0000B
IE	Interrupt Enable	A8H	EA	EPCA_LVD	EADC_SPI	ES	ET1	EX1	ET0	EX0	0000 0000B
IP	Interrupt Priority Low	B8H	-	PPCA_LVD	PADC_LVD	PS	PT1	PX1	PT0	PX0	x000 0000B
IPH	Interrupt Priority High	B7H	-	PPCA_LVDH	PADC_LVDH	PSH	PT1H	PX1H	PT0H	PX0H	x000 0000B
SADEN	Slave Address Mask	B9H									0000 0000B
SADDR	Slave Address	A9H									0000 0000B
WAKE_CLKO	CLK_Output Power down Wake-up control register	8FH	PCAWAKEUP	RxD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CLKO	T0CLKO	0000 xx00B

1. Serial Port 1 (UART1) Control Register: SCON and PCON

Serial port 1 of STC12C2052AD series has two control registers: Serial port control register (SCON) and PCON which used to select Baud-Rate

SCON: Serial port Control Register (Bit-Addressable)

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

FE: Framing Error bit. The SMOD0 bit must be set to enable access to the FE bit

0: The FE bit is not cleared by valid frames but should be cleared by software.

1: This bit set by the receiver when an invalid stop bit id detected.

SM0,SM1 : Serial Port Mode Bit 0/1.

SM0	SM1	Description	Baud rate
0	0	8-bit shift register	SYSclk/12
0	1	8-bit UART	variable
1	0	9-bit UART	SYSclk/64 or SYSclk/32(SMOD=1)
1	1	9-bit UART	variable

SM2 : Enable the automatic address recognition feature in mode 2 and 3. If SM2=1, RI will not be set unless the received 9th data bit is 1, indicating an address, and the received byte is a Given or Broadcast address. In mode1, if SM2=1 then RI will not be set unless a valid stop Bit was received, and the received byte is a Given or Broadcast address. In mode 0, SM2 should be 0.

REN : When set enables serial reception.

TB8 : The 9th data bit which will be transmitted in mode 2 and 3.

RB8 : In mode 2 and 3, the received 9th data bit will go into this bit.

TI : Transmit interrupt flag. Set by hardware when a byte of data has been transmitted by UART0 (after the 8th bit in 8-bit UART Mode, or at the beginning of the STOP bit in 9-bit UART Mode). When the UART0 interrupt is enabled, setting this bit causes the CPU to vector to the UART0 interrupt service routine. This bit must be cleared manually by software.

RI : Receive interrupt flag. Set to '1' by hardware when a byte of data has been received by UART0 (set at the STOP bit sam-pling time). When the UART0 interrupt is enabled, setting this bit to '1' causes the CPU to vector to the UART0 interrupt service routine. This bit must be cleared manually by software.

SMOD/PCON.7 in PCON register can be used to set whether the baud rates of mode 1, mode2 and mode 3 are doubled or not.

PCON: Power Control register (Non bit-addressable)

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

SMOD: double Baud rate control bit.

0 : Disable double Baud rate of the UART.

1 : Enable double Baud rate of the UART in mode 1,2,or 3.

SMOD0: Frame Error select.

0 : SCON.7 is SM0 function.

1 : SCON.7 is FE function. Note that FE will be set after a frame error regardless of the state of SMOD0.

2. SBUF: Serial port 1 Data Buffer register (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SBUF	99H	name								

It is used as the buffer register in transmission and reception.The serial port buffer register (SBUF) is really two buffers. Writing to SBUF loads data to be transmitted, and reading SBUF accesses received data. These are two separate and distinct registers, the transimit write-only register, and the receive read-only register.

3. AUXR: Auxiliary register (Non bit-addressable)

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

T0x12 : Timer 0 clock source bit.
- 0 : The clock source of Timer 0 is SYSclk/12. It will compatible to the traditional 80C51 MCU
- 1 : The clock source of Timer 0 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

T1x12 : Timer 1 clock source bit.
- 0 : The clock source of Timer 1 is SYSclk/12. It will compatible to the traditional 80C51 MCU
- 1 : The clock source of Timer 1 is SYSclk/1. It will drive the T0 faster than a traditional 80C51 MCU

UART_M0x6 : Baud rate select bit of UART1 while it is working under Mode-0
- 0 : The baud-rate of UART in mode 0 is SYSclk/12.
- 1 : The baud-rate of UART in mode 0 is SYSclk/2.

EADCI : Enable/Disable interrupt from A/D converter
- 0 : (default) Inhibit the ADC functional block to generate interrupt to the MCU
- 1 : Enable the ADC functional block to generate interrupt to the MCU

ESPI : Enable/Disable interrupt from Serial Peripheral Interface (SPI)
- 0 : (default)Inhibit the SPI functional block to generate interrupt to the MCU
- 1 : Enable the SPI functional block to generate interrupt to the MCU

ELVDI : Enable/Disable interrupt from low-voltage sensor
- 0 : (default)Inhibit the low-voltage sensor functional block to generate interrupt to the MCU
- 1 : Enable the low-voltage sensor functional block to generate interrupt to the MCU

4. Slave Address Control registers SADEN and SADDR

SADEN: Slave Address Mask register
SADDR: Slave Address register
SADDR register is combined with SADEN register to form Given/Broadcast Address for automatic address recognition. In fact, SADEN function as the "mask" register for SADDR register. The following is the example for it.

```
SADDR   =  1100 0000
SADEN   =  1111 1101
Given   =  1100 00x0  ──►  The Given slave address will be checked except bit 1 is
                           treated as "don't care".
```

The Broadcast Address for each slave is created by taking the logical OR of SADDR and SADEN. Zero in this result is considered as "don't care" and a Broad cast Address of all " don't care". This disables the automatic address detection feature.

5. Power down wake-up register: WAKE_CLKO (Non bit-Addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
WAKE_CLKO	8FH	name	PCAWAKEUP	RXD_PIN_IE	T1_PIN_IE	T0_PIN_IE	-	-	T1CKLO	T0CKLO

PCAWAKEUP: When set and the associated-PCA interrupt control registers is configured correctly, the CEXn pin of PCA function is enabled to wake up MCU from power-down state.

RXD_PIN_IE: When set and the associated-UART interrupt control registers is configured correctly, the RXD pin (P3.0) is enabled to wake up MCU from power-down state.

T1_PIN_IE : When set and the associated-Timer1 interrupt control registers is configured correctly, the T1 pin (P3.5) is enabled to wake up MCU from power-down state.

T0_PIN_IE : When set and the associated-Timer0 interrupt control registers is configured correctly, the T1 pin (P3.4) is enabled to wake up MCU from power-down state.

T1CKLO : When set, P3.5 is enabled to be the clock output of Timer 1. The clock rate is Timer 1overflow rate divided by 2.

T0CKLO : When set, P3.4 is enabled to be the clock output of Timer 0. The clock rate is Timer 0overflow rate divided by 2.

6. Registers related with UART1 interrupt : IE, IP and IPH

IE: Interrupt Enable Rsgister (Bit-addressable)

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

EA : disables all interrupts.
If EA = 0,no interrupt will be acknowledged.
If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

ES : Serial port 1(UART1) interrupt enable bit.
If ES = 0, Serial port (UART) interrupt will be diabled.
If ES = 1, Serial port (UART) interrupt is enabled.0

IPH: Interrupt Priority High Register (Non bit-addressable)

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

IP: Interrupt Priority Register (Bit-addressable)

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

PSH, PS: Serial Port (UART) interrupt priority control bits.
if PSH=0 and PS=0, UART interrupt is assigned lowest priority (priority 0).
if PSH=0 and PS=1, UART interrupt is assigned lower priority (priority 1).
if PSH=1 and PS=0, UART interrupt is assigned higher priority (priority 2).
if PSH=1 and PS=1, UART interrupt is assigned highest priority (priority 3).

8.2 UART Operation Modes

The serial port (UART) can be operated in 4 different modes which are configured by setting SM0 and SM1 in SFR SCON. Mode 1, Mode 2 and Mode 3 are asynchronous communication. In Mode 0, UART is used as a simple shift register.

8.2.1 Mode 0: 8-Bit Shift Register

Mode 0, selected by writing 0s into bits SM1 and SM0 of SCON, puts the serial port into 8-bit shift register mode. Serial data enters and exits through RXD. TXD outputs the shift clock. Eight data bits are transmitted/received with the least-significant (LSB) first. The baud rate is fixed at 1/12 the System clock cycle in the default state. If AUXR.5(UART_M0x6) is set, the baud rate is 1/2 System clock cycle.

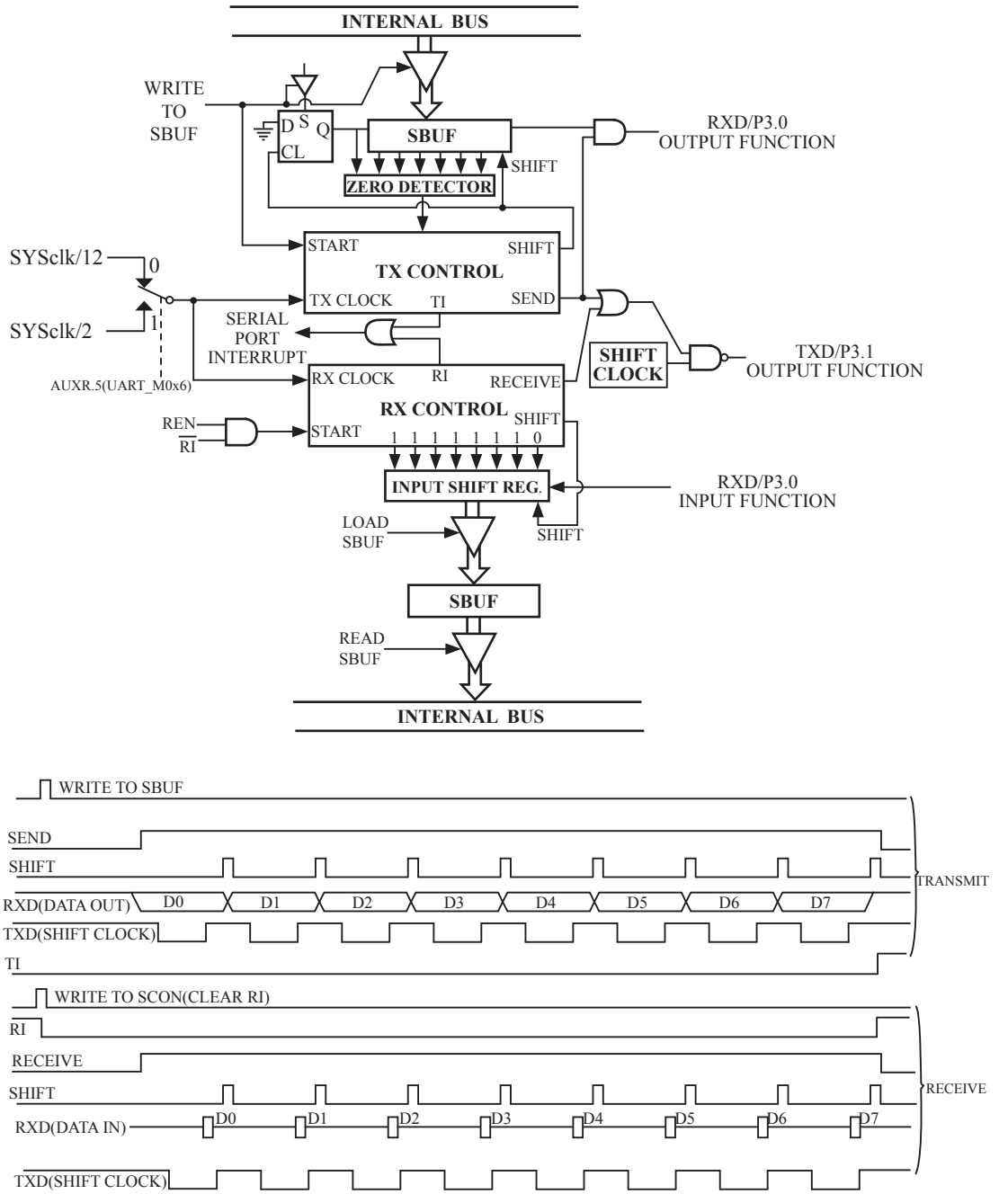
Transmission is initiated by any instruction that uses SBUF as a destination register. The “write to SBUF” signal also loads a “1” into the 9th position of the transmit shift register and tells the TX Control block to commence a transmission. The internal timing is such that one full system clock cycle will elapse between "write to SBUF," and activation of SEND.

SEND transfers the output of the shift register to the alternate output function line of P3.0, and also transfers Shift Clock to the alternate output function line of P3.1. At the falling edge of the Shift Clock, the contents of the shift register are shifted one position to the right.

As data bits shift out to the right, “0” come in from the left. When the MSB of the data byte is at the output position of the shift register, then the “1” that was initially loaded into the 9th position is just to the left of the MSB, and all positions to the left of that contains zeroes. This condition flags the TX Control block to do one last shift and then deactivate SEND and set TI. Both of these actions occur after "write to SBUF".

Reception is initiated by the condition REN=1 and RI=0. After that, the RX Control unit writes the bits 11111110 to the receive shift register, and in the next clock phase activates RECEIVE. RECEIVE enables SHIFT CLOCK to the alternate output function line of P3.1.At RECEIVE is active, the contents of the receive shift register are shifted to the left one position. The value that comes in from the right is the value that was sampled at the P3.0 pin the rising edge of Shift clock.

As data bits come in from the right, “1”s shift out to the left. When the “0” that was initially loaded into the right-most position arrives at the left-most position in the shift register, it flags the RX Control block to do one last shift and load SBUF. Then RECEIVE is cleared and RI is set.



Serial Port Mode 0

8.2.2 Mode 1: 8-Bit UART with Variable Baud Rate

10 bits are transmitted through TXD or received through RXD. The frame data includes a start bit(0), 8 data bits and a stop bit(1). One receive, the stop bit goes into RB8 in SFR – SCON. The baud rate is determined by the Timer 1 or BRT overflow rate.

Baud rate in mode 1 = $(2^{SMOD}/32)$ x Timer 1 overflow rate
When T1x12=0, Timer 1 overflow rate = SYSclk/12/(256-TH1);
When T1x12=1, Timer 1 overflow rate = SYSclk / (256-TH1);

Transmission is initiated by any instruction that uses SBUF as a destination register. The "write to SBUF" signal also loads a "1" into the 9th bit position of the transmit shift register and flags the TX Control unit that a transmission is requested. Transmission actually happens at the next rollover of divided-by-16 counter. Thus the bit times are synchronized to the divided-by-16 counter, not to the "write to SBUF" signal.

The transmission begins with activation of $\overline{SEND}$, which puts the start bit at TXD. One bit time later, DATA is activated, which enables the output bit of the transmit shift register to TXD. The first shift pulse occurs one bit time after that.

As data bits shift out to the right, zeroes are clocked in from the left. When the MSB of the data byte is at the output position of the shift register, then the 1 that was initially loaded into the 9th position is just to the left of the MSB, and all positions to the left of that contain zeroes. This condition flags the TX Control unit to do one last shift and then deactivate $\overline{SEND}$ and set TI. This occurs at the 10th divide-by-16 rollover after "write to SBUF."

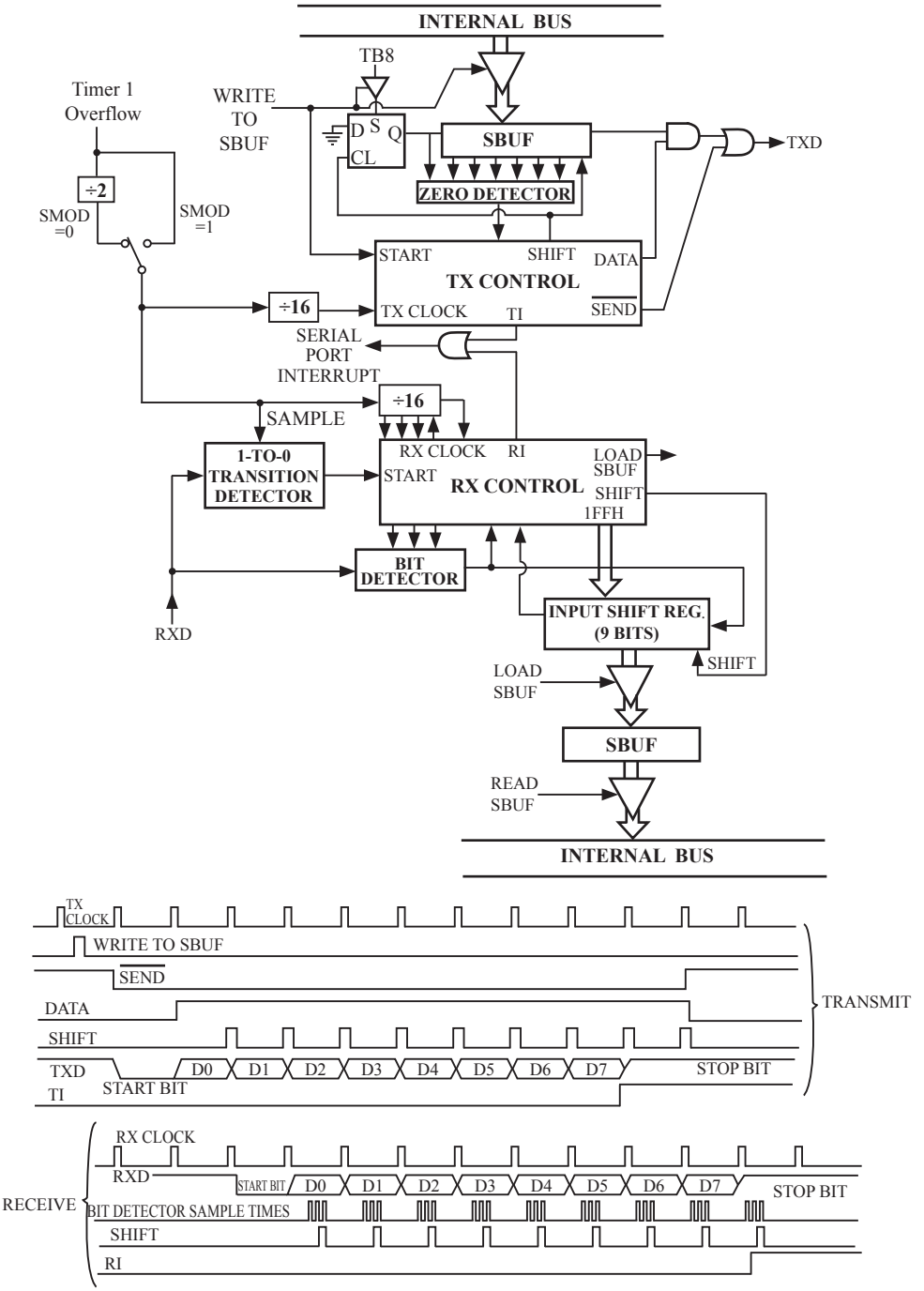
Reception is initiated by a 1-to-0 transition detected at RXD. For this purpose, RXD is sampled at a rate of 16 times the established baud rate. When a transition is detected, the divided-by-16 counter is immediately reset, and 1FFH is written into the input shift register. Resetting the divided-by-16 counter aligns its roll-overs with the boundaries of the incoming bit times.

The 16 states of the counter divide each bit time into 16ths. At the 7th, 8th and 9th counter states of each bit time, the bit detector samples the value of RXD. The value accepted is the value that was seen in at least 2 of the 3 samples. This is done to reject noise. In order to reject false bits, if the value accepted during the first bit time is not a 0, the receive circuits are reset and the unit continues looking for another 1-to-0 transition. This is to provide rejection of false start bits. If the start bit is valid, it is shifted into the input shift register, and reception of the rest of the frame proceeds.

As data bits come in from the right, "1"s shift out to the left. When the start bit arrives at the left most position in the shift register,(which is a 9-bit register in Mode 1), it flags the RX Control block to do one last shift, load SBUF and RB8, and set RI. The signal to load SBUF and RB8 and to set RI is generated if, and only if, the following conditions are met at the time the final shift pulse is generated.

1) RI=0 and
2) Either SM2=0, or the received stop bit = 1

If either of these two conditions is not met, the received frame is irretrievably lost. If both conditions are met, the stop bit goes into RB8, the 8 data bits go into SBUF, and RI is activated. At this time, whether or not the above conditions are met, the unit continues looking for a 1-to-0 transition in RXD.



Serial Port Mode 1

8.2.3 Mode 2: 9-Bit UART with Fixed Baud Rate

11 bits are transmitted through TXD or received through RXD. The frame data includes a start bit(0), 8 data bits, a programmable 9th data bit and a stop bit(1). On transmit, the 9th data bit comes from TB8 in SCON. On receive, the 9th data bit goes into RB8 in SCON. The baud rate is programmable to either 1/32 or 1/64 the System clock cycle.

Baud rate in mode 2 = $(2^{SMOD}/64)$ x SYSclk

Transmission is initiated by any instruction that uses SBUF as a destination register. The "write to SBUF" signal also loads TB8 into the 9th bit position of the transmit shift register and flags the TX Control unit that a transmission is requested. Transmission actually happens at the next rollover of divided-by-16 counter. Thus the bit times are synchronized to the divided-by-16 counter, not to the "write to SBUF" signal.

The transmission begins when /SEND is activated, which puts the start bit at TXD. One bit time later, DATA is activated, which enables the output bit of the transmit shift register to TXD. The first shift pulse occurs one bit time after that. The first shift clocks a "1"(the stop bit) into the 9th bit position on the shift register. Thereafter, only "0"s are clocked in. As data bits shift out to the right, "0"s are clocked in from the left. When TB8 of the data byte is at the output position of the shift register, then the stop bit is just to the left of TB8, and all positions to the left of that contains "0"s. This condition flags the TX Control unit to do one last shift, then deactivate /SEND and set TI. This occurs at the 11th divided-by-16 rollover after "write to SBUF".

Reception is initiated by a 1-to-0 transition detected at RXD. For this purpose, RXD is sampled at a rate of 16 times whatever baud rate has been estabished. When a transition is detected, the divided-by-16 counter is immediately reset, and 1FFH is written into the input shift register.

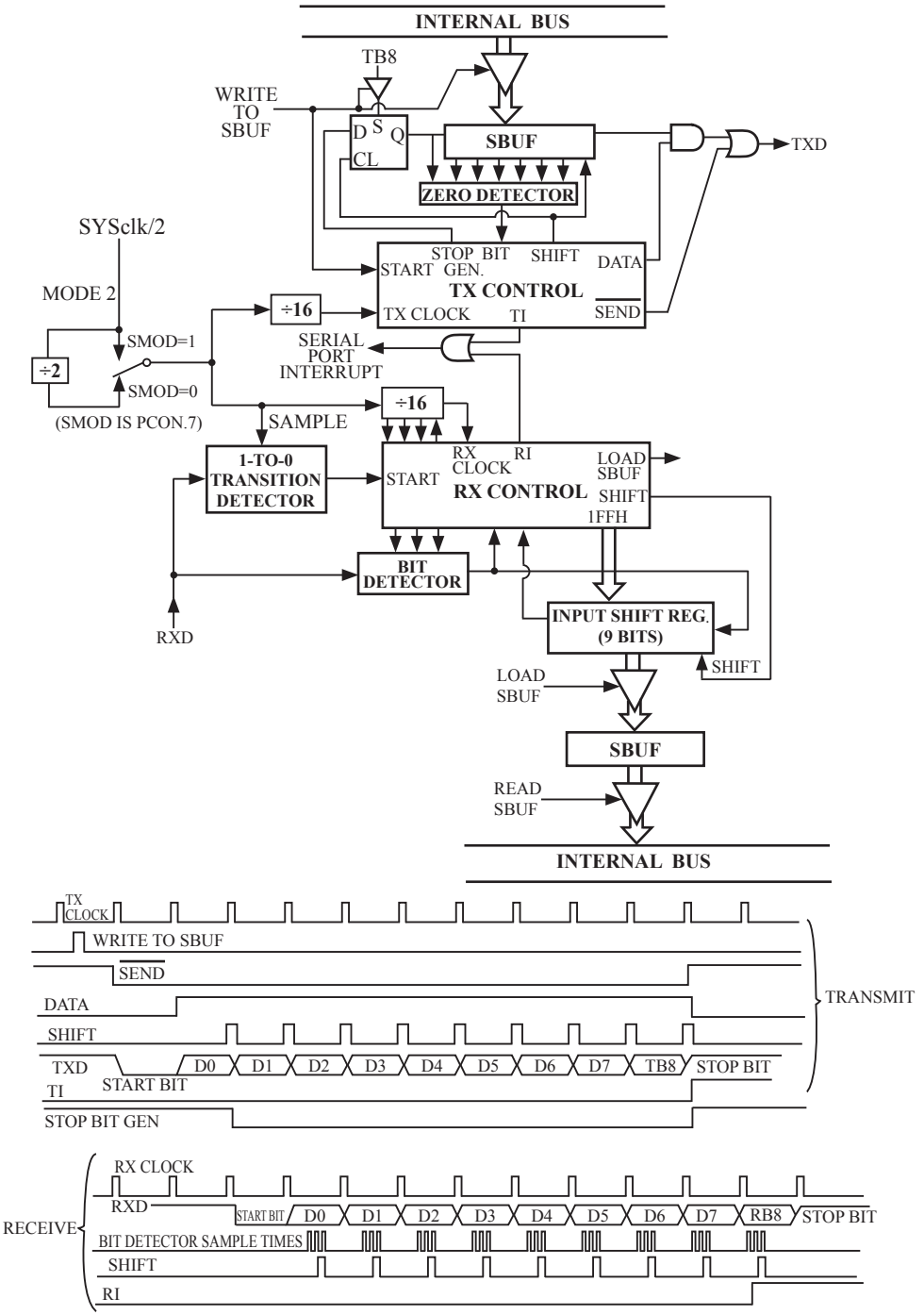
At the 7th, 8th and 9th counter states of each bit time, the bit detector samples the value of RXD. The value accepted is the value that was seen in at least 2 of the 3 samples. This is done to reject noise. In order to reject false bits, if the value accepted during the first bit time is not a 0, the receive circuits are reset and the unit continues looking for another 1-to-0 transition. If the start bit is valid, it is shifted into the input shift register, and reception of the rest of the frame proceeds.

As data bits come in from the right, "1"s shift out to the left. When the start bit arrives at the leftmost position in the shift register,(which is a 9-bit register in Mode-2 and 3), it flags the RX Control block to do one last shift, load SBUF and RB8, and set RI. The signal to load SBUF and RB8 and to set RI is generated if, and only if, the following conditions are met at the time the final shift pulse is generated.:

1) RI=0 and
2) Either SM2=0, or the received 9th data bit = 1

If either of these two conditions is not met, the received frame is irretrievably lost. If both conditions are met, the stop bit goes into RB8, the first 8 data bits go into SBUF, and RI is activated. At this time, whether or not the above conditions are met, the unit continues looking for a 1-to-0 transition at the RXD input.

Note that the value of received stop bit is irrelevant to SBUF, RB8 or RI.



Serial Port Mode 2

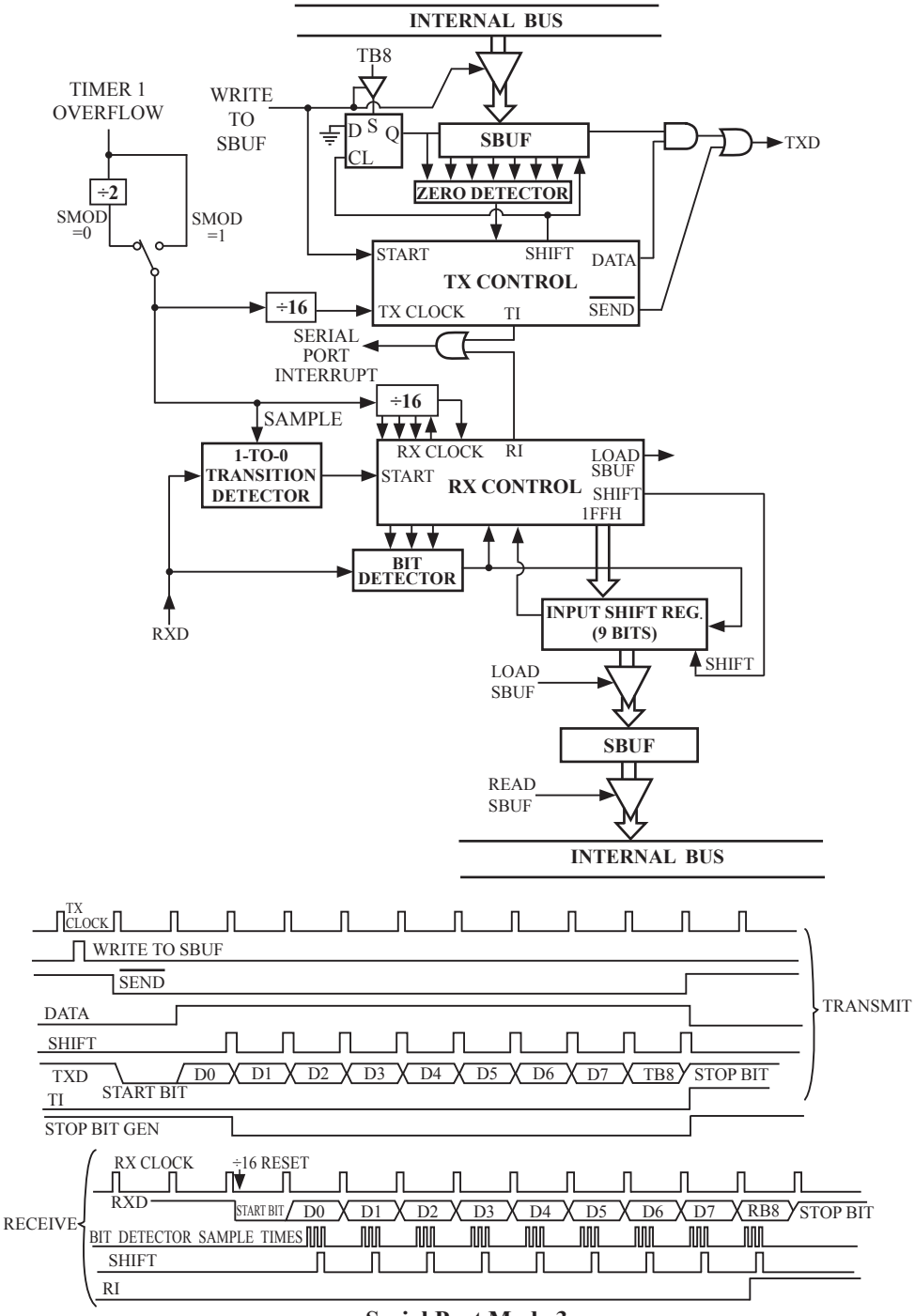
8.2.4 Mode 3: 9-Bit UART with Variable Baud Rate

Mode 3 is the same as mode 2 except the baud rate is variable.

Baud rate in mode 3 = $(2^{SMOD}/32)$ x Timer 1 overflow rate

When T1x12=0, Timer 1 overflow rate = SYSclk/12/(256-TH1);
When T1x12=1, Timer 1 overflow rate = SYSclk / (256-TH1);

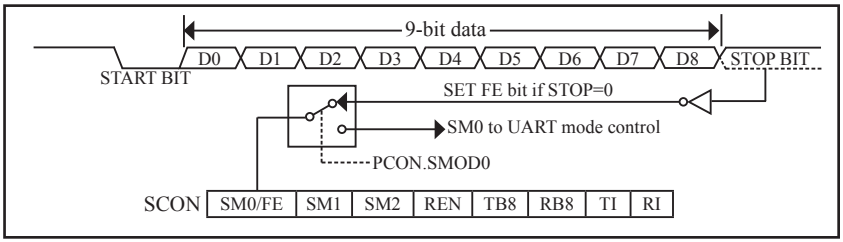
In all four modes, transmission is initiated by any instruction that use SBUF as a destination register. Reception is initiated in mode 0 by the condition RI = 0 and REN = 1. Reception is initiated in the other modes by the incoming start bit with 1-to-0 transition if REN=1.



Serial Port Mode 3

8.3 Frame Error Detection

When used for frame error detect, the UART looks for missing stop bits in the communication. A missing bit will set the FE bit in the SCON register. The FE bit shares the SCON.7 bit with SM0 and the function of SCON.7 is determined by PCON.6(SMOD0). If SMOD0 is set then SCON.7 functions as FE. SCON.7 functions as SM0 when SMOD0 is cleared.When used as FE,SCON.7 can only be cleared by software.Refer to the following figure.



UART Frame Error Detection

8.4 Multiprocessor Communications

Modes 2 and 3 have a special provision for multiproceasor communications. In these modes 9 data bits are received.The 9th one goes into RB8. Then comes a stop bit. The port can be programmed such that when the stop bit is received,the serial port interrupt will be activated only if RB8 = 1. This feature is enabled by setting bit SM2 in SCON. A way to use this feature in multiprocessor systems is as follows.

When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave.An address byte differs from a data byte in that the 9th bit is 1 in an address byte and 0 in a data byte.With SM2 = 1, no slave will be interrupted by a data byte. An address byte, however,will interrupt all slaves, so that each slave can examine the received byte and see if it is being addressed.The addressed slave will clear its SM2 bit and prepare to receive the data bytes that will be coming. The slaves that weren't being addressed leave their SM2s set and go on about their business, ignoring the coming data bytes.

SM2 has no effect in Mode 0,and in Mode 1 can be used to check the validity of the stop bit. In a Mode 1 reception, if SM2 = 1, the receive interrupt will not be activated unless a vatid stop bit is received.

8.5 Automatic Address Recognition

Automatic Address Recognition is a future which allows the UART to recognize certain addresses in the serial bit stream by using hardware to make the comparisons. This feature saves a great deal of software overhead by eliminating the need for the software to examine every serial address which passes by the serial port. This feature is enabled by setting the SM2 bit in SCON. In the 9-bit UART modes, Mode 2 and Mode 3, the Receive interrupt flag(RI) will be automatically set when the received byte contains either the "Given" address or the "Broadcast" address. The 9-bit mode requires that the 9th information bit is a "1" to indicate that the received information is an address and not data.

The 8-bit mode is called Mode 1. In this mode the RI flag will be set if SM2 is enabled and the information received has a valid stop bit following the 8 address bits and the information is either a Given or Broadcast address.

Mode 0 is the Shift Register mode and SM2 is ignored.

Using the Automatic Address Recognition feature allows a master to selectively communicate with one or more slaves by invoking the given slave address or addresses. All of the slaves may be contacted by using the broadcast address. Two special function registers are used to define the slave's address, SADDR, and the address mask, SADEN. SADEN is used to define which bits in the SADDR are to be used and which bits are "don't care". The SADEN mask can be logically ANDed with the SADDR to create the "Given" address which the master will use for addressing each of the slaves. Use of the Given address allows multiple slaves to be recognized which excluding others. The following examples will help to show the versatility of this scheme :

Slave 0 SADDR = 1100 0000
SADEN = 1111 1101
GIVEN = 1100 00x0

Slave 1 SADDR = 1100 0000
SADEN = 1111 1110
GIVEN = 1100 000x

In the previous example SADDR is the same and the SADEN data is used to differentiate between the two slaves. Slave 0 requires a "0" in bit 0 and it ignores bit 1. Slave 1 requires a "0" in bit 1 and bit 0 is ignored. A unique address for slave 0 would be 11000010 since slave 1 requires a "0" in bit 1. A unique address for slave 1 would be 11000001 since a "1" in bit 0 will exclude slave 0. Both slaves can be selected at the same time by an address which has bit 0=0 (for slave 0) and bit 1 =0 (for salve 1). Thus, both could be addressed with 11000000.

In a more complex system the following could be used to select slaves 1 and 2 while excluding slave 0:

Slave 0 SADDR = 1100 0000
SADEN = 1111 1001
GIVEN = 1100 0xx0

Slave 1 SADDR = 1110 0000
SADEN = 1111 1010
GIVEN = 1110 0x0x

Slave 2 SADDR = 1110 0000
SADEN = 1111 1100
GIVEN = 1110 00xx

In the above example the differentiation among the 3 slaves is in the lower 3 address bits.Slave 0 requires that bit0 = 0 and it can be uniquely addressed by 11100110. Slave 1 requires that bit 1=0 and it can be uniquely addressed by 11100101. Slave 2 requires that bit 2=0 and its unique address is 11100011. To select Salve 0 and 1 and exclude Slave 2, use address 11100100, since it is necessary to make bit2=1 to exclude Slave 2.

The Broadcast Address for each slave is created by taking the logic OR of SADDR and SADEN. Zeros in this result are trended as don't cares. In most cares, interpreting the don't cares as ones, the broadcast address will be FF hexadecimal.

Upon reset SADDR and SADEN are loaded with "0"s. This produces a given address of all "don't cares as well as a Broadcast address of all "don't cares". This effectively disables the Automatic Addressing mode and allows the microcontroller to use standard 80C51-type UART drivers which do not make use of this feature.

Example: write an program that continually transmits characters from a transmit buffer. If incoming characters are detected on the serial port, store them in the receive buffer starting at internal RAM location 50H. Assume that the STC12C2052AD series MCU serial port has already been initialized in mode 1.

Solution:

```
        ORG     0030H
        MOV     R0,     #30H            ;pointer for tx buffer
        MOV     R1,     #50H            ;pointer for rx buffer
LOOP:   JB      RI,     RECEIVE         ;character received?
                                        ;yes: process it
        JB      TI,     TX              ;previous character transmitted ?
                                        ;yes: process it
        SJMP    LOOP                    ;no: continue checking
TX:     MOV     A,      @R0             ;get character from tx buffer
        MOV     C,      P               ;put parity bit in C
        CPL     C                       ;change to odd parity
        MOV     ACC.7,  C               ;add to character code
        CLR     TI                      ;clear transmit flag
        MOV     SBUF,   A               ;send character
        CLR     ACC.7                   ;strip off parity bit
        INC     R0                      ;point to next character in buffer
        CJNE    R0,     #50H,   LOOP    ;end of buffer?
                                        ;no: continue
        MOV     R0,     #30H            ;yes: recycle
        SJMP    LOOP                    ;continue checking
RX:     CLR     RI                      ;clear receive flag
        MOV     A,      SBUF            ;read character into A
        MOV     C,      P               ;for odd parity in A, P should be set
        CPL     C                       ;complementing correctly indicates "error"
        CLR     ACC.7                   ;strip off parity
        MOV     @R1,    A               ;store received character in buffer
        INC     R1                      ;point to next location in buffer
        SJMP    LOOP                    ;continue checking
        END
```

8.6 Buad Rates

The baud rate in Mode 0 is fixed:

$$\text{Mode 0 Baud Rate} = \frac{\text{SYSclk}}{12} \quad \text{when AUXR.5/UART\_M0x6 =0}$$

$$\text{or} = \frac{\text{SYSclk}}{2} \quad \text{when AUXR.5/UART\_M0x6 =1}$$

The baud rate in Mode 2 depends on the value of bit SMOD in Special Function Register PCON. If SMOD =0 (which is the value on reset), the baud rate $^1/_{64}$ the System clock cycle. If SMOD = 1, the baud rate is $^1/_{32}$ the System clock cycle .

$$\text{Mode 2 Baud Rate} = \frac{2^{SMOD}}{64} \times (\text{SYSclk})$$

In the STC12C2052AD, the baud rates in Modes 1 and 3 are determined by Timer 1 overflow rate.
The baud rate in Mode 1 and 3 are fixed:

$$\text{Mode 1,3 Baud rate} = (2^{SMOD}/32) \times \text{timer 1 overflow rate}$$

Timer 1 overflow rate = (SYSclk/12)/(256 - TH1);

When Timer 1 is used as the baud rate generator, the Timer 1 interrupt should be disabled in this application. The Timer itself can be configured for either "timer" or "cormter" operation, and in any of its 3 running modes. In the most typcial applications, it is configured for "timer" operation, in the auto-reload mode (high nibble of TMOD = 0010B).
One can achieve very low baud rate with Timer 1 by leaving the Timer 1 interrupt enabled, and configuring the Timer to run as a 16-bit timer (high nibble of TMOD = 0001B), and using the Timer 1 interrupt to do a l6-bit software reload.

The following figure lists various commonly used baud rates and how they can be obtained from Timer 1.

Baud Rate	f_{osc}	SMOD	Timer 1		
			C/$\overline{T}$	Mode	Reload Value
Mode 0 MAX:1MHZ	12MHZ	X	X	X	X
Mode 2 MAX:375K	12MHZ	1	X	X	X
Mode 1,3:62.5K	12MHZ	1	0	2	FFH
19.2K	11.059MHZ	1	0	2	FDH
9.6K	11.059MHZ	0	0	2	FDH
4.8K	11.059MHZ	0	0	2	FAH
2.4K	11.059MHZ	0	0	2	F4H
1.2K	11.059MHZ	0	0	2	E8H
137.5	11.986MHZ	0	0	2	1DH
110	6MHZ	0	0	2	72H
110	12MHZ	0	0	1	FEEBH

Timer 1 Generated Commonly Used Baud Rates

8.7 Demo Programs about UART (C and ASM)

1. C program:

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC12C5Axx Series MCU UART (8-bit/9-bit)Demo ----------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

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

typedef   unsigned char       BYTE;
typedef   unsigned int        WORD;

#define   FOSC    18432000L                       //System frequency
#define   BAUD    9600                            //UART baudrate

/*Define UART parity mode*/
#define   NONE_PARITY     0                       //None parity
#define   ODD_PARITY      1                       //Odd parity
#define   EVEN_PARITY     2                       //Even parity
#define   MARK_PARITY     3                       //Mark parity
#define   SPACE_PARITY    4                       //Space parity

#define   PARITYBIT       EVEN_PARITY             //Testing even parity

sbit      bit9 = P2^2;                            //P2.2 show UART data bit9
bit       busy;

void SendData(BYTE dat);
void SendString(char *s);

void main()
{
#if (PARITYBIT == NONE_PARITY)
          SCON = 0x50;                            //8-bit variable UART
#elif (PARITYBIT == ODD_PARITY) || (PARITYBIT == EVEN_PARITY) || (PARITYBIT == MARK_PARITY)
          SCON = 0xda;                            //9-bit variable UART, parity bit initial to 1
#elif (PARITYBIT == SPACE_PARITY)
          SCON = 0xd2;                            //9-bit variable UART, parity bit initial to 0
#endif
```

```
	TMOD	= 0x20;				//Set Timer1 as 8-bit auto reload mode
	TH1	= TL1 = -(FOSC/12/32/BAUD);	//Set auto-reload vaule
	TR1	= 1;				//Timer1 start run
	ES	= 1;				//Enable UART interrupt
	EA	= 1;				//Open master interrupt switch

	SendString("STC12C2052AD\r\nUart Test !\r\n");
	while(1);
}

/*----------------------------
UART interrupt service routine
----------------------------*/
void Uart_Isr() interrupt 4 using 1
{
	if (RI)
	{
		RI = 0;				//Clear receive interrupt flag
		P0 = SBUF;			//P0 show UART data
		bit9 = RB8;			//P2.2 show parity bit
	}
	if (TI)
	{
		TI = 0;				//Clear transmit interrupt flag
		busy = 0;			//Clear transmit busy flag
	}
}
/*----------------------------
Send a byte data to UART
Input: dat (data to be sent)
Output:None
----------------------------*/
void SendData(BYTE dat)
{
	while (busy);				//Wait for the completion of the previous data is sent
	ACC = dat;				//Calculate the even parity bit P (PSW.0)
	if (P)					//Set the parity bit according to P
	{
	#if (PARITYBIT == ODD_PARITY)
		TB8 = 0;			//Set parity bit to 0
	#elif (PARITYBIT == EVEN_PARITY)
		TB8 = 1;			//Set parity bit to 1
	#endif
	}
```

```
        else
        {
        #if (PARITYBIT == ODD_PARITY)
                TB8 = 1;                                    //Set parity bit to 1
        #elif (PARITYBIT == EVEN_PARITY)
                TB8 = 0;                                    //Set parity bit to 0
        #endif
        }
                busy = 1;
                SBUF = ACC;                                 //Send data to UART buffer
}

/*----------------------------
Send a string to UART
Input: s (address of string)
Output:None
----------------------------*/
void SendString(char *s)
{
        while (*s)                                          //Check the end of the string
        {
                SendData(*s++);                             //Send current char and increment string ptr
        }
}
```

2. Assembly program:

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ------------------------------------*/
/* --- STC12C5Axx Series MCU UART (8-bit/9-bit)Demo ----------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

;/*Define UART parity mode*/
#define   NONE_PARITY   0                       //None parity
#define   ODD_PARITY    1                       //Odd parity
#define   EVEN_PARITY   2                       //Even parity
#define   MARK_PARITY   3                       //Mark parity
#define   SPACE_PARITY  4                       //Space parity

#define   PARITYBIT     EVEN_PARITY             //Testing even parity
;-----------------------------------------
BUSY    BIT     20H.0                           ;transmit busy flag
;-----------------------------------------
        ORG     0000H
        LJMP    MAIN

        ORG     0023H
        LJMP    UART_ISR
;-----------------------------------------
        ORG     0100H
MAIN:
        CLR     BUSY
        CLR     EA
        MOV     SP,     #3FH
#if (PARITYBIT == NONE_PARITY)
        MOV     SCON,   #50H                    ;8-bit variable UART
#elif (PARITYBIT == ODD_PARITY) || (PARITYBIT == EVEN_PARITY) || (PARITYBIT == MARK_PARITY)
        MOV     SCON,   #0DAH                   ;9-bit variable UART, parity bit initial to 1
#elif (PARITYBIT == SPACE_PARITY)
        MOV     SCON,   #0D2H                   ;9-bit variable UART, parity bit initial to 0
#endif
;--------------------------------
```

```
        MOV     TMOD,   #20H                    ;Set Timer1 as 8-bit auto reload mode
        MOV     A,      #0FBH                   ;256-18432000/12/32/9600
        MOV     TH1,    A                       ;Set auto-reload vaule
        MOV     TL1,    A
        SETB    TR1                             ;Timer1 start run
        SETB    ES                              ;Enable UART interrupt
        SETB    EA                              ;Open master interrupt switch
;-------------------------------
        MOV     DPTR,   #TESTSTR                ;Load string address to DPTR
        LCALL   SENDSTRING                      ;Send string
;-------------------------------
        SJMP    $
;-----------------------------------------
TESTSTR:                    ;Test string
         DB     "STC12C2052AD Uart Test !",     0DH,0AH,0

;/*----------------------------
;UART2 interrupt service routine
;----------------------------*/
UART_ISR:
        PUSH    ACC
        PUSH    PSW
        JNB     RI,     CHECKTI                 ;Check RI bit
        CLR     RI                              ;Clear RI bit
        MOV     P0,     SBUF                    ;P0 show UART data
        MOV     C,      RB8
        MOV     P2.2,   C                       ;P2.2 show parity bit
CHECKTI:
        JNB     TI,     ISR_EXIT                ;Check S2TI bit
        CLR     TI                              ;Clear S2TI bit
        CLR     BUSY                            ;Clear transmit busy flag
ISR_EXIT:
        POP     PSW
        POP     ACC
        RETI

;/*----------------------------
;Send a byte data to UART
;Input: ACC (data to be sent)
;Output:None
;----------------------------*/
SENDDATA:
        JB      BUSY,   $                       ;Wait for the completion of the previous data is sent
        MOV     ACC,    A                       ;Calculate the even parity bit P (PSW.0)
        JNB     P,      EVEN1INACC              ;Set the parity bit according to P
```

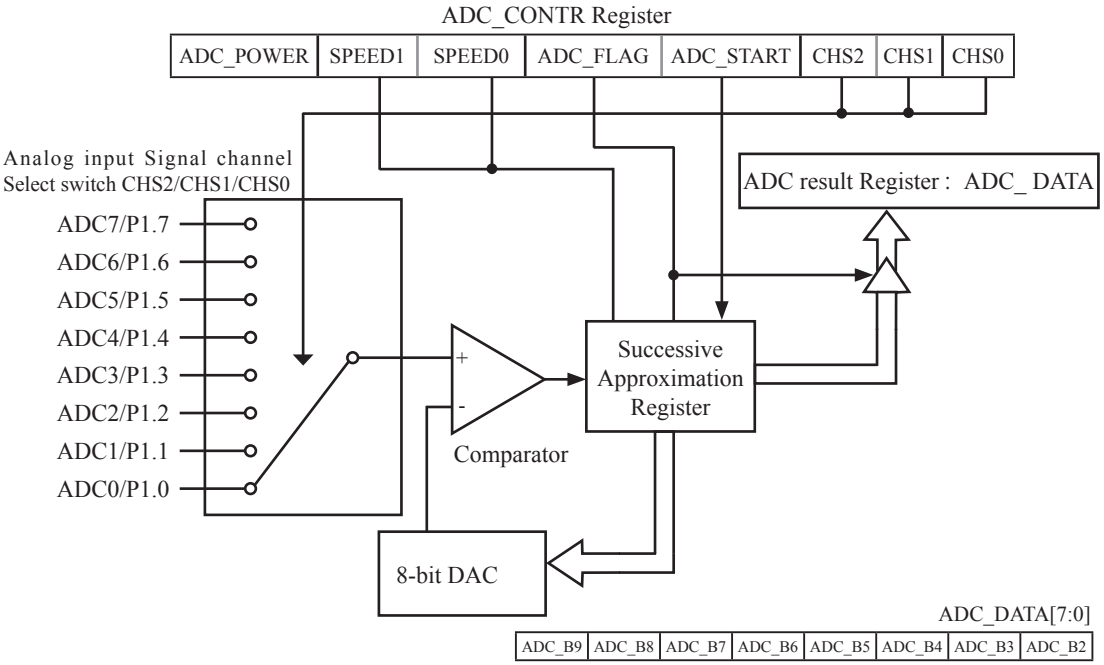
```
ODD1INACC:
#if (PARITYBIT == ODD_PARITY)
        CLR     TB8                             ;Set parity bit to 0
#elif (PARITYBIT == EVEN_PARITY)
        SETB    TB8                             ;Set parity bit to 1
#endif
        SJMP    PARITYBITOK
EVEN1INACC:
#if (PARITYBIT == ODD_PARITY)
        SETB    TB8                             ;Set parity bit to 1
#elif (PARITYBIT == EVEN_PARITY)
        CLR     TB8                             ;Set parity bit to 0
#endif
PARITYBITOK:                                    ;Parity bit set completed
        SETB    BUSY
        MOV     SBUF,   A                       ;Send data to UART buffer
        RET

;/*----------------------------
;Send a string to UART
;Input: DPTR (address of string)
;Output:None
;----------------------------*/
SENDSTRING:
        CLR     A
        MOVC    A,      @A+DPTR                 ;Get current char
        JZ      STRINGEND                       ;Check the end of the string
        INC     DPTR                            ;increment string ptr
        LCALL   SENDDATA                        ;Send current char
        SJMP    SENDSTRING                      ;Check next
STRINGEND:
        RET
;------------------------------------------
        END
```

Chapter 9. Analog to Digital Converter

9.1 A/D Converter Structure

STC12C2052AD series MCU with A/D conversion function have 8-channel and 8-bit high-speed A/D converter whose speed is up to 100KHz (100 thousand times per second). the 8-channel ADC, which are on P1 port (P1.0-P1.7) , can be used as temperature detection, battery voltage detection, key scan, spectrum detection, etc. After power on reset, P1 ports are in weak pull-up mode. Users can set any one of 8 channels as A/D conversion through software. And those I/O ports not as ADC function can continue to be used as I/O ports.

STC12C2052AD series MCU ADC (A/D converter) structure is shown below.



The ADC on STC12C2052AD is an 8-bit resolution, successive-approximation approach, medium-speed A/D converter. V_{REFP}/V_{REFM} is the positive/negative reference voltage input for internal voltage-scaling DAC use, the typical sink current on it is 600uA ~ 1mA. For STC12C2052AD, these two references are internally tied to VCC and GND separately.

Conversion is invoked since ADC_STRAT(ADC_CONTR.3) bit is set. Before invoking conversion, ADC_POWER/ADC_CONTR.7 bit should be set first in order to turn on the power of analog front-end in ADC circuitry. Prior to ADC conversion, the desired I/O ports for analog inputs should be configured as input-only or open-drain mode first. The converter takes around a fourth cycles to sample analog input data and other three fourths cycles in successive-approximation steps. Total conversion time is controlled by two register bits – SPEED1 and SPEED0. Eight analog channels are available on P1 and only one of them is connected to to the comparator depending on the selection bits {CHS2,CHS1,CHS0}. When conversion is completed, the result will be saved onto ADC_DATA register. After the result are completed and saved, ADC_FLAG is also set. ADC_FLAG should be cleared in software. The ADC interrupt service routine vectors to 2Bh . When the chip enters idle mode or power-down mode, the power of ADC is gated off by hardware.

Calculating the result according to the following formula:

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

In the above formula, Vin stand for analog input channel voltage, Vcc stand for actual operation voltage

9.2 Registers for ADC

Mnemonic	Description	Address	bit address and Symbol MSB LSB	Reset value
P1M0	P1 Configuration 0	91H		0000 0000B
P1M1	P1 Configuration 1	92H		0000 0000B
ADC_CONTR	ADC Control Register	C5H	ADC_POWER | SPEED1 | SPEED0 | ADC_FLAG | ADC_START | CHS2 | CHS1 | CHS0	0000 0000B
ADC_DATA	ADC Result Register	C6H		0000 0000B
IE	Interrupt Enable	A8H	EA | EPCA_LVD | EADC_SPI | ES | ET1 | EX1 | ET0 | EX0	0000 0000B
IP	Interrupt Priority Low	B8H	- | PPCA_LVD | PADC_SPI | PS | PT1 | PX1 | PT0 | PX0	x000 0000B
IPH	Interrupt Priority High	B7H	- | PPCA_LVDH | PADC_SPIH | PSH | PT1H | PX1H | PT0H | PX0H	x000 0000B
AUXR	Auxiliary register	A2H	T0x12 | T1x12 | UART_M0x12 | EADCI | ESPI | ELVDI | - | -	0000 00xxB

1. P1 Analog Function Configure register: P1M0 and P1M1

Those P1 ports which need to be used as A/D converter should be first set in open-drain or high-impedance (input-only) mode through registers P1M0 and P1M1.

P1 Configure <P1.7, P1.6, P1.5, P1.4, P1.3, P1.2, P1.1, P1.0 port> (P1 address：90H)

P1M0[7 : 0]	P1M1 [7 : 0]	I/O ports Mode
0	0	quasi_bidirectional(standard 8051 I/O port output） , Sink Current up to 20mA , pull-up Current is 230μA , Because of manufactured error, the actual pull-up current is 250uA ~ 150uA
0	1	push-pull output(strong pull-up output，current can be up to 20mA, resistors need to be added to restrict current
1	0	input-only (high-impedance). If any P1 port need to be used as ADC, its mode is optional between input-only (high-impedance) and open-drain mode.
1	1	Open Drain，internal pull-up resistors should be disabled and external pull-up resistors need to join. If any P1 port need to be used as ADC, its mode is optional between input-only (high-impedance) and open-drain mode.

Example: MOV P1M0, #10100000B
MOV P1M1, #11000000B
;P1.7 in Open Drain mode, P1.6 in strong push-pull output, P1.5 in high-impedance input, P1.4/P1.3/P1.2/P1.1/P1.0 in quasi_bidirectional/weak pull-up

2. ADC control register: ADC_CONTR (Non bit-addressable)

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

When operating to ADC_CONTR register, "MOV" should be used, while "AND" and "OR" don not be recommended to use

ADC_POWER : When clear shut down the power of ADC block. When set turn on the power of ADC block.
SPEED1, SPEED0 : Conversion speed selection.

SPEED1	SPEED0	Times needed by an A/D Coversion
0	0	1080 clock cycles are needed for a conversion.
0	1	810 clock cycles are needed for a conversion.
1	0	540 clock cycles are needed for a conversion.
1	1	270 clock cycles are needed for a conversion. When the CPU operation frequency is 27MHz, the speed of ADC is about 100KHz.

The clock source used by ADC block of STC12C2052AD series MCU is On-chip R/C clock which is not divided by Clock divider register CLK_DIV.

ADC_FLAG : ADC interrupt flag.It will be set by the device after the device has finished a conversion, and should be cleared by the user's software.

ADC_STRAT : ADC start bit, which enable ADC conversion.It will automatically cleared by the device after the device has finished the conversion.

CHS2 ~ CHS0 : Used to select one analog input source from 8 channels.

CHS2	CHS1	CHS0	Source
0	0	0	P1.0 (default) as the A/D channel input
0	0	1	P1.1 as the A/D channel input
0	1	0	P1.2 as the A/D channel input
0	1	1	P1.3 as the A/D channel input
1	0	0	P1.4 as the A/D channel input
1	0	1	P1.5 as the A/D channel input
1	1	0	P1.6 as the A/D channel input
1	1	1	P1.7 as the A/D channel input

Note : The corresponding bits in P1ASF should be configured correctly before starting A/D conversion. The sepecific P1ASF bits should be set corresponding with the desired channels.

Because it will by delayed 4 CPU clocks after the instruction which set ADC_CONTR register has been executed, Four "NOP" instructions should be added after setting ADC_CONTR register. See the following code:

```
MOV     ADC_CONTR,      #DATA
NOP
NOP
NOP
NOP
MOV     A,          ADC_CONTR
                        ;Only delayed 4 clocks, can the ADC_CONTR be read correctly.
```

3. ADC result register: ADC_DATA

ADC_DATA is used to save the ADC result, their format as shown below:

Mnemonic	Add	Name	B7	B6	B5	B4	B3	B2	B1	B0
ADC_DATA	C6h	ADC result register high								

Calculating the result according to the following formula:

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

In the above formula, Vin stand for analog input channel voltage, Vcc stand for actual operation voltage

4. Registers related with UART1 interrupt : IE, AUXR, IP and IPH

IE: Interrupt Enable Rsgister (Bit-addressable)

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

EA : disables all interrupts.
If EA = 0,no interrupt will be acknowledged.
If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

EADC_SPI : Interrupt controller of Serial Peripheral Interface (SPI) and A/D Converter (ADC).
If EADC_SPI = 0, Disable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).
If EADC_SPI = 1, Enable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).

AUXR: Auxiliary register (Non bit-addressable)

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

EADCI : Enable/Disable interrupt from A/D converter
0 : (default) Inhibit the ADC functional block to generate interrupt to the MCU
1 : Enable the ADC functional block to generate interrupt to the MCU

IPH: Interrupt Priority High Register (Non bit-addressable)

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

IP: Interrupt Priority Register (Bit-addressable)

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

PADC_SPIH, PADC_SPI : Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt priority control bits.

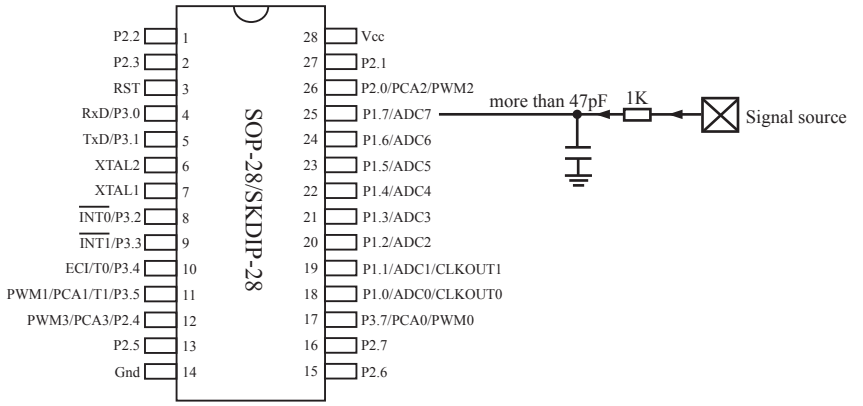
if PADC_SPIH=0 and PADC_SPI=0, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 0).

if PADC_SPIH=0 and PADC_SPI=1, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 1).

if PADC_SPIH=1 and PADC_SPI=0, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 2).

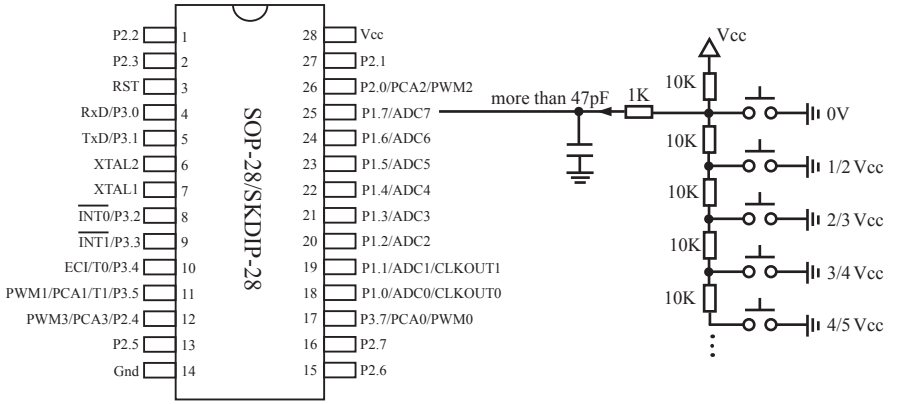
if PADC_SPIH=1 and PADC_SPI=1, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 3).

9.3 Application Circuit of A/D Converter

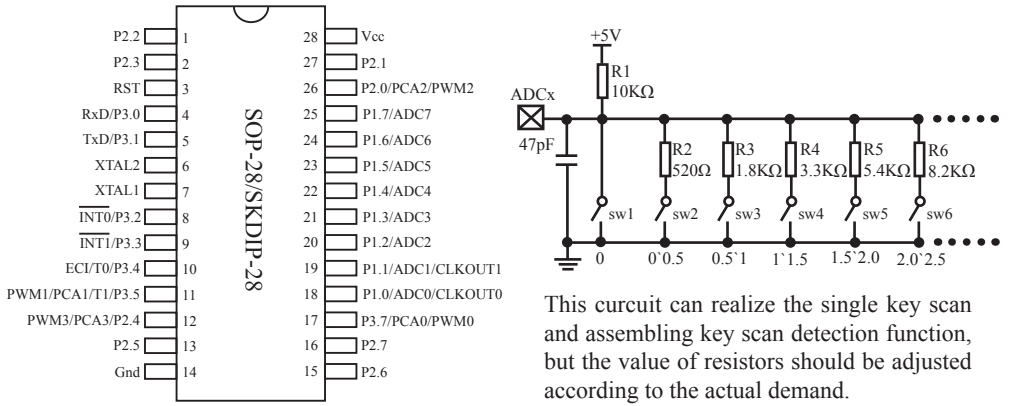


ADC function in P1 port, P1.0 - P1.7 in all 8 channels

9.4 ADC Application Circuit for Key Scan

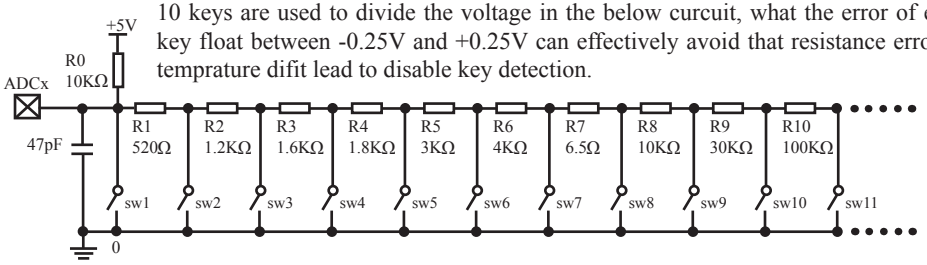


ADC function in P1 port, P1.0 - P1.7 in all 8 channels



This curcuit can realize the single key scan and assembling key scan detection function, but the value of resistors should be adjusted according to the actual demand.

10 keys are used to divide the voltage in the below curcuit, what the error of each key float between -0.25V and +0.25V can effectively avoid that resistance error or temprature difit lead to disable key detection.



9.5 A/D reference voltage source

STC12C2052AD series ADC reference voltage is from MCU power supply voltage directly, so it can work without an external reference voltage source. If the required precision is relatively high, then you maybe using a stable reference voltage source, in order to calculate the operating voltage VCC, then calculate the ADC exact value. For example, you can connect a 1.25V(or 1.00V, ect. …) to ADC channel 7, according to the conversion result, you can get the actual VCC voltage, thus you can calculate other 7 channels ADC results. (Vcc is constant in short time)

9.6 Program using interrupts to demostrate A/D Conversion

There are two example procedures using interrupts to demostrate A/D conversion, one written in C langugage and the other in assembly language.

1. C language code listing:

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series MCU A/D Conversion Demo -------------------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

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

#define   FOSC    18432000L
#define   BAUD    9600

typedef   unsigned char       BYTE;
typedef   unsigned int        WORD;

/*Declare SFR associated with the ADC */
sfr AUXR          =   0x8e;
sfr ADC_CONTR     =   0xC5;                 //ADC control register
sfr ADC_DATA      =   0xC6;                 //ADC high 8-bit result register
sfr ADC_LOW2      =   0xBE;                 //ADC low 2-bit result register
sfr P1M0          =   0x91;                 //P1 mode control register0
sfr P1M1          =   0x92;                 //P1 mode control register1

/*Define ADC operation const for ADC_CONTR*/
#define   ADC_POWER     0x80                //ADC power control bit
#define   ADC_FLAG      0x10                //ADC complete flag
#define   ADC_START     0x08                //ADC start control bit
#define   ADC_SPEEDLL   0x00                //1080 clocks
#define   ADC_SPEEDL    0x20                //810 clocks
#define   ADC_SPEEDH    0x40                //540 clocks
#define   ADC_SPEEDHH   0x60                //270 clocks

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

```
BYTE ch = 0;                    //ADC channel NO.

void main()
{
    InitUart();                 //Init UART, use to show ADC result
    InitADC();                  //Init ADC sfr
    AUXR |= 0x10;               //set EADCI
    IE = 0xa0;                  //Enable ADC interrupt and Open master interrupt switch
                                //Start A/D conversion
    while (1);
}

/*------------------------------
ADC interrupt service routine
------------------------------*/
void adc_isr() interrupt 5 using 1
{
    ADC_CONTR &= !ADC_FLAG;             //Clear ADC interrupt flag

    SendData(ch);                       //Show Channel NO.
    SendData(ADC_DATA);                 //Get ADC high 8-bit result and Send to UART

    //if you want show 10-bit result, uncomment next line
    //  SendData(ADC_LOW2);             //Show ADC low 2-bit result

    if (++ch > 7) ch = 0;               //switch to next channel
    ADC_CONTR = ADC_POWER | ADC_SPEEDLL | ADC_START | ch;
}

/*------------------------------
Initial ADC sfr
------------------------------*/
void InitADC()
{
    P1 = P1M0 = P1M1 = 0xff;            //Set all P1 as Open-Drain mode
    ADC_DATA = 0;                       //Clear previous result
    ADC_CONTR = ADC_POWER | ADC_SPEEDLL | ADC_START | ch;
    Delay(2);                           //ADC power-on delay and Start A/D conversion
}

/*------------------------------
Initial UART
------------------------------*/
```

```
void InitUart()
{
	SCON = 0x5a;				//8 bit data ,no parity bit
	TMOD = 0x20;				//T1 as 8-bit auto reload
	TH1 = TL1 = -(FOSC/12/32/BAUD);		//Set Uart baudrate
	TR1 = 1;				//T1 start running
}

/*----------------------------
Send one byte data to PC
Input: dat (UART data)
Output:-
----------------------------*/
void SendData(BYTE dat)
{
	while (!TI);				//Wait for the previous data is sent
	TI = 0;					//Clear TI flag
	SBUF = dat;				//Send current data
}

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

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

2. Assembly language code listing:

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------*/
;/* --- STC 1T Series MCU A/D Conversion Demo -------------------------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

;/*Declare SFR associated with the ADC */
AUXR        EQU   8EH
ADC_CONTR   EQU   0C5H          ;ADC control register
ADC_DATA    EQU   0C6H          ;ADC high 8-bit result register
ADC_LOW2    EQU   0BEH          ;ADC low 2-bit result register
P1M0        EQU   091H          ;P1 mode control register0
P1M1        EQU   092H          ;P1 mode control register1

;/*Define ADC operation const for ADC_CONTR*/
ADC_POWER   EQU   80H           ;ADC power control bit
ADC_FLAG    EQU   10H           ;ADC complete flag
ADC_START   EQU   08H           ;ADC start control bit
ADC_SPEEDLL EQU   00H           ;1080 clocks
ADC_SPEEDL  EQU    20H          ;810 clocks
ADC_SPEEDH  EQU   40H           ;540 clocks
ADC_SPEEDHH EQU   60H           ;270 clocks

ADCCH       DATA  20H           ;ADC channel NO.

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

        ORG   002BH
        LJMP  ADC_ISR
;----------------------------------------
        ORG   0100H
MAIN:
        MOV   SP,           #3FH
        MOV   ADCCH, #0
        LCALL INIT_UART                 ;Init UART, use to show ADC result
        LCALL INIT_ADC                  ;Init ADC sfr
        ORL   AUXR,   #10H              ;set EADCI
        MOV   IE,     #0A0H     ;Enable ADC interrupt and Open master interrupt switch
        SJMP  $
```

```
;/*----------------------------
;ADC interrupt service routine
;----------------------------*/
ADC_ISR:
        PUSH    ACC
        PUSH    PSW

        ANL     ADC_CONTR,      #NOT ADC_FLAG           ;Clear ADC interrupt flag
        MOV     A,          ADCCH
        LCALL   SEND_DATA                               ;Send channel NO.
        MOV     A,          ADC_DATA                    ;Get ADC high 8-bit result
        LCALL   SEND_DATA                               ;Send to UART

;//if you want show 10-bit result, uncomment next 2 lines
;       MOV     A,          ADC_LOW2                    ;Get ADC low 2-bit result
;       LCALL   SEND_DATA                               ;Send to UART

        INC     ADCCH
        MOV     A,          ADCCH
        ANL     A,          #07H
        MOV     ADCCH, A
        ORL     A,          #ADC_POWER | ADC_SPEEDLL | ADC_START
        MOV     ADC_CONTR,A                     ;ADC power-on delay and re-start A/D conversion
        POP     PSW
        POP     ACC
        RETI

;/*----------------------------
;Initial ADC sfr
;----------------------------*/
INIT_ADC:
        MOV     A,          #0FFH
        MOV     P1,         A
        MOV     P1M0,       A
        MOV     P1M1,       A                           ;Set all P1 as Open-Drain mode
        MOV     ADC_DATA,       #0                      ;Clear previous result
        MOV     A,          ADCCH
        ORL     A,          #ADC_POWER | ADC_SPEEDLL | ADC_START
        MOV     ADC_CONTR,      A                   ;ADC power-on delay and Start A/D conversion
        MOV     A,#2
        LCALL   DELAY
        RET
```

```
;/*----------------------------
;Initial UART
;----------------------------*/
INIT_UART:
        MOV     SCON,   #5AH            ;8 bit data ,no parity bit
        MOV     TMOD,   #20H            ;T1 as 8-bit auto reload
        MOV     A,      #-5             ;Set Uart baudrate -(18432000/12/32/9600)
        MOV     TH1,    A               ;Set T1 reload value
        MOV     TL1,    A
        SETB    TR1                     ;T1 start running
        RET

;/*----------------------------
;Send one byte data to PC
;Input: ACC (UART data)
;Output:-
;----------------------------*/
SEND_DATA:
        JNB     TI,     $               ;Wait for the previous data is sent
        CLR     TI                      ;Clear TI flag
        MOV     SBUF,   A               ;Send current data
        RET

;/*----------------------------
;Software delay function
;----------------------------*/
DELAY:
        MOV     R2,     A
        CLR     A
        MOV     R0,     A
        MOV     R1,     A
DELAY1:
        DJNZ    R0,     DELAY1
        DJNZ    R1,     DELAY1
        DJNZ    R2,     DELAY1
        RET

        END
```

9.7 Program using polling to demostrate A/D Conversion

There are two example procedures using inquiry to demostrate A/D conversion, one written in C langugage and the other in assembly language.

1. C language code listing:

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited -------------------------------------*/
/* --- STC 1T Series MCU A/D Conversion Demo ----------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

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

#define   FOSC    18432000L
#define   BAUD    9600

typedef   unsigned char     BYTE;
typedef   unsigned int      WORD;

/*Declare SFR associated with the ADC */
sfr       ADC_CONTR    =  0xC5;          //ADC control register
sfr       ADC_DATA     =  0xC6;          //ADC high 8-bit result register
sfr       ADC_LOW2     =  0xBE;          //ADC low 2-bit result register
sfr       P1M0         =  0x91;          //P1 mode control register0
sfr       P1M1         =  0x92;          //P1 mode control register1

/*Define ADC operation const for ADC_CONTR*/
#define   ADC_POWER    0x80              //ADC power control bit
#define   ADC_FLAG     0x10              //ADC complete flag
#define   ADC_START    0x08              //ADC start control bit
#define   ADC_SPEEDLL  0x00              //1080 clocks
#define   ADC_SPEEDL   0x20              //810 clocks
#define   ADC_SPEEDH   0x40              //540 clocks
#define   ADC_SPEEDHH  0x60              //270 clocks

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

```
void main()
{
	InitUart();					//Init UART, use to show ADC result
	InitADC();					//Init ADC sfr
	while (1)
	{
		ShowResult(0);			//Show Channel0
		ShowResult(1);			//Show Channel1
		ShowResult(2);			//Show Channel2
		ShowResult(3);			//Show Channel3
		ShowResult(4);			//Show Channel4
		ShowResult(5);			//Show Channel5
		ShowResult(6);			//Show Channel6
		ShowResult(7);			//Show Channel7
	}
}

/*----------------------------
Send ADC result to UART
----------------------------*/
void ShowResult(BYTE ch)
{
	SendData(ch);				//Show Channel NO.
	SendData(GetADCResult(ch));		//Show ADC high 8-bit result

//if you want show 10-bit result, uncomment next line
//	SendData(ADC_LOW2);			//Show ADC low 2-bit result
}

/*----------------------------
Get ADC result
----------------------------*/
BYTE GetADCResult(BYTE ch)
{
	ADC_CONTR = ADC_POWER | ADC_SPEEDLL | ch | ADC_START;
	_nop_();				//Must wait before inquiry
	_nop_();
	_nop_();
	_nop_();
	while (!(ADC_CONTR & ADC_FLAG));	//Wait complete flag
	ADC_CONTR &= ~ADC_FLAG;			//Close ADC

	return ADC_DATA;				//Return ADC result
}
```

```
/*----------------------------
Initial UART
----------------------------*/
void InitUart()
{
	SCON = 0x5a;				//8 bit data ,no parity bit
	TMOD = 0x20;				//T1 as 8-bit auto reload
	TH1 = TL1 = -(FOSC/12/32/BAUD);		//Set Uart baudrate
	TR1 = 1;				//T1 start running
}

/*----------------------------
Initial ADC sfr
----------------------------*/
void InitADC()
{
	 P1 = P1M0 = P1M1 = 0xff;		//Set all P1 as Open-Drain mode
	ADC_DATA = 0;				//Clear previous result
	ADC_CONTR = ADC_POWER | ADC_SPEEDLL;
	Delay(2);				//ADC power-on and delay
}

/*----------------------------
Send one byte data to PC
Input: dat (UART data)
Output:-
----------------------------*/
void SendData(BYTE dat)
{
	while (!TI);				//Wait for the previous data is sent
	TI = 0;					//Clear TI flag
	SBUF = dat;				//Send current data
}

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

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

2. Assembly language code listing:

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------*/
;/* --- STC 1T Series MCU A/D Conversion Demo ---------------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

;/*Declare SFR associated with the ADC */
ADC_CONTR       EQU     0C5H            ;ADC control register
ADC_DATA        EQU     0C6H            ;ADC high 8-bit result register
ADC_LOW2        EQU     0BEH            ;ADC low 2-bit result register
P1M0            EQU     091H            ;P1 mode control register0
P1M1            EQU     092H            ;P1 mode control register1

;/*Define ADC operation const for ADC_CONTR*/
ADC_POWER       EQU     80H             ;ADC power control bit
ADC_FLAG        EQU     10H             ;ADC complete flag
ADC_START       EQU     08H             ;ADC start control bit
ADC_SPEEDLL     EQU     00H             ;1080 clocks
ADC_SPEEDL      EQU     20H             ;810 clocks
ADC_SPEEDH      EQU     40H             ;540 clocks
ADC_SPEEDHH     EQU     60H             ;270 clocks

;------------------------------------------
        ORG     0000H
        LJMP    MAIN
;------------------------------------------
        ORG     0100H
MAIN:
        LCALL   INIT_UART               ;Init UART, use to show ADC result
        LCALL   INIT_ADC                ;Init ADC sfr
;------------------------------
NEXT:
        MOV     A,      #0
        LCALL   SHOW_RESULT             ;Show channel0 result
        MOV     A,      #1
        LCALL   SHOW_RESULT             ;Show channel1 result
        MOV     A,      #2
        LCALL   SHOW_RESULT             ;Show channel2 result
```

```
        MOV     A,      #3
        LCALL   SHOW_RESULT             ;Show channel3 result
        MOV     A,      #4
        LCALL   SHOW_RESULT             ;Show channel4 result
        MOV     A,      #5
        LCALL   SHOW_RESULT             ;Show channel5 result
        MOV     A,      #6
        LCALL   SHOW_RESULT             ;Show channel6 result
        MOV     A,      #7
        LCALL   SHOW_RESULT             ;Show channel7 result

        SJMP    NEXT

;/*----------------------------
;Send ADC result to UART
;Input: ACC (ADC channel NO.)
;Output:-
;----------------------------*/
SHOW_RESULT:
        LCALL   SEND_DATA               ;Show Channel NO.
        LCALL   GET_ADC_RESULT          ;Get high 8-bit ADC result
        LCALL   SEND_DATA               ;Show result

;//if you want show 10-bit result, uncomment next 2 lines
;       MOV     A,      ADC_LOW2        ;Get low 2-bit ADC result
;       LCALL   SEND_DATA               ;Show result
        RET

;/*----------------------------
;Read ADC conversion result
;Input: ACC (ADC channel NO.)
;Output:ACC (ADC result)
;----------------------------*/
GET_ADC_RESULT:
        ORL     A,      #ADC_POWER | ADC_SPEEDLL | ADC_START
        MOV     ADC_CONTR,      A       ;Start A/D conversion
        NOP                             ;Must wait before inquiry
        NOP
        NOP
        NOP
WAIT:
        MOV     A,      ADC_CONTR                       ;Wait complete flag
        JNB     ACC.4,  WAIT    ;ADC_FLAG(ADC_CONTR.4)
        ANL     ADC_CONTR       ,#NOT ADC_FLAG          ;Clear ADC_FLAG
        MOV     A,      ADC_DATA                                ;Return ADC result
        RET
```

```
;/*----------------------------
;Initial ADC sfr
;----------------------------*/
INIT_ADC:
        MOV     A,      #0FFH
        MOV     P1,     A
        MOV     P1M0,   A
        MOV     P1M1,   A                       ;Set all P1 as Open-Drain mode
        MOV     ADC_DATA,#0                     ;Clear previous result
        MOV     ADC_CONTR,      #ADC_POWER | ADC_SPEEDLL
        MOV     A,      #2                      ;ADC power-on and delay
        LCALL   DELAY
        RET
;/*----------------------------
;Initial UART
;----------------------------*/
INIT_UART:
        MOV     SCON,   #5AH                    ;8 bit data ,no parity bit
        MOV     TMOD,   #20H                    ;T1 as 8-bit auto reload
        MOV     A,      #-5                     ;Set Uart baudrate -(18432000/12/32/9600)
        MOV     TH1,    A                       ;Set T1 reload value
        MOV     TL1,    A
        SETB    TR1                             ;T1 start running
        RET

;/*----------------------------
;Send one byte data to PC
;Input: ACC (UART data)
;Output:-
;----------------------------*/
SEND_DATA:
        JNB     TI,$                            ;Wait for the previous data is sent
        CLR     TI                              ;Clear TI flag
        MOV     SBUF,   A                       ;Send current data
        RET

;/*----------------------------
;Software delay function
;----------------------------*/
DELAY:
        MOV     R2,     A
        CLR     A
        MOV     R0,     A
        MOV     R1,     A
DELAY1:
        DJNZ    R0,     DELAY1
        DJNZ    R1,     DELAY1
        DJNZ    R2,     DELAY1
        RET

        END
```

Chapter 10. Programmable Counter Array(PCA)

The Programmable Counter Array is a special 16-bit Timer that has four 16-bit capture/compare modules associated with it. Each of the modules can be programmed to operate in one of four modes: rising and/or falling edge capture (calculator of duty length for high/low pulse), software timer, high-speed output, or pulse width modulator. Module 0 is connected to pin P3.7, module 1 to pin P3.5, module 2 to pin P2.0, module 3 to pin P2.4 in STC12C2052AD series.

The PCA timer is a common time base for all four modules and can be programmed to run at 1/12 the system clock, 1/2 the system clock, the Timer 0 overflow or the input on ECI pin(P3.4). The timer count source is determined from CPS1 and CPS0 bits in the CMOD SFR.

10.1 SFRs related with PCA

PCA/PWM SFRs table

Mnemonic	Description	Add	Bit address and Symbol								Reset Value
			B7	B6	B5	B4	B3	B2	B1	B0	
CCON	PCA Control Register	D8H	CF	CR	-	-	-	-	CCF1	CCF0	00xx,xx00
CMOD	PCA Mode Register	D9H	CIDL	-	-	-	-	CPS1	CPS0	ECF	0xxx,x000
CCAPM0	PCA Module 0 Mode Register	DAH	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0	x000,0000
CCAPM1	PCA Module 1 Mode Register	DBH	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1	x000,0000
CL	PCA Base Timer Low	E9H									0000,0000
CH	PCA Base Timer High	F9H									0000,0000
CCAP0L	PCA Module-0 Capture Register Low	EAH									0000,0000
CCAP0H	PCA Module-0 Capture Register High	FAH									0000,0000
CCAP1L	PCA Module-1 Capture Register Low	EBH									0000,0000
CCAP1H	PCA Module-1 Capture Register High	FBH									0000,0000
PCA_PWM0	PCA PWM Mode Auxiliary Register 0	F2H	-	-	-	-	-	-	EPC0H	EPC0L	xxxx,xx00
PCA_PWM1	PCA PWM Mode Auxiliary Register 1	F3H	-	-	-	-	-	-	EPC1H	EPC1L	xxxx,xx00

1. PCA operation mode register: CMOD (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CMOD	D9H	name	CIDL	-	-	-	-	CPS1	CPS0	ECF

CIDL : PCA Counter control bit in Idle mode.
If CIDL=0, the PCA counter will continue functioning during idle mode.
If CIDL=1, the PCA counter will be gated off during idle mode.

CPS1, CPS0 : PCA Counter Pulse source Select bits.

CPS1	CPS0	Select PCA/PWM clock source
0	0	0, System clock/12, SYSclk/12
0	1	1, System clock/2, SYSclk/2
1	0	2, Timer 0 overflow. PCA/PWM clock can up to SYSclk because Timer 0 can operate in 1T mode. Frequency-adjustable PWM output can be achieved by changing the Timer 0 overflow.
1	1	3, Exrenal clock from ECI/P3.4 pin (max speed = SYSclk/2)

For example, If CPS1/CPS0=0/0, PCA/PWM clock source is SYSclk/12.
If users need to select SYSclk/3 as PCA clock source, Timer 0 should be set to operate in 1T mode and generate an overflow every 3 counting pulse.

ECF : PCA Counter Overflow interrupt Enable bit.
ECF=0 disables CF bit in CCON to generate an interrupt.
ECF=1 enables CF bit in CCON to generate an interrupt.

2. PCA control register : CCON (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CCON	D8H	name	CF	CR	-	-	-	-	CCF1	CCF0

CF : PCA Counter overflow flag. Set by hardware when the counter rolls over. CF flags an interrupt if bit ECF in CMOD is set. CF may be set by either hardware or software but can only be cleared by software.

CR : PCA Counter Run control bit. Set by software to turn the PCA counter on. Must be cleared by software to turn the PCA counter off.

CCF1 : PCA Module 1 interrupt flag. Set by hardware when a match or capture from module 1 occurs. Must be cleared by software. A match means the value of the PCA counter equals the value of the Capture/Compare register in module 1. A capture means a specific edge from PCA1 happens, so the Capture/Compare register latches the value of the PCA counter, and the CCF1 is set.

CCF0 : PCA Module 0 interrupt flag. Set by hardware when a match or capture from module 0 occurs. Must be cleared by software. A match means the value of the PCA counter equals the value of the Capture/Compare register in module 0. A capture means a specific edge from PCA0 happens, so the Capture/Compare register latches the value of the PCA counter, and the CCF0 is set.

3. PCA capture/campare register CCAPM0 and CCAPM1

Each module in the PCA has a special function register associated with it. These registers are CCAPMn, n=0 and 1. CCAPM0 for module 0, CCAPM1 for module 1. The register contains the bits that control the mode in which each module will operate. The ECCFn bit enables the CCFn flag in the CCON SFR to generate an interrupt when a match or compare occurs in the associated module. PWMn enables the pulse width modulation mode. The TOGn bit when set causes the CCPn output associated with the module to toggle when there is a match between the PCA counter and the module's capture/compare register. The match bit(MATn) when set will cause the CCFn bit in the CCON register to be set when there is a match between the PCA counter and the module's capture/compare register.

The next two bits CAPNn and CAPPn determine the edge that a capture input will be active on. The CAPNn bit enables the negative edge, and the CAPPn bit enables the positive edge. If both bits are set, both edges will be enabled and a capture will occur for either transition. The bit ECOMn when set enables the comparator function.

Capture/Compare register of PCA module 0 : CCAPM0 (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CCAPM0	DAH	name	-	ECOM0	CAPP0	CAPN0	MAT0	TOG0	PWM0	ECCF0

ECOM0 : Comparator Enable bit.
ECOM0=0 disables the comparator function;
ECOM0=1 enables the comparator function.

CAPP0 : Capture Positive control bit.
CAPP0=1 enables positive edge capture.

CAPN0 : Capture Negative control bit.
CAPN0=1 enables negative edge capture.

MAT0 : Match control bit.
When MAT0 = 1, a match of the PCA counter with this module's compare/capture register causes the CCF0 bit in CCON to be set.

TOG0 : Toggle control bit.
When TOG0=1, a match of the PCA counter with this module's compare/capture register causes the CCP0 pin to toggle.
(PCA0/PWM0/P3.7)

PWM0 : Pulse Width Modulation.
PWM0=1 enables the CCP0 pin to be used as a pulse width modulated output.
(PCA0/PWM0/P3.7)

ECCF0 : Enable CCF0 interrupt.
Enables compare/capture flag CCF0 in the CCON register to generate an interrupt.

Capture/Compare register of PCA module 1 : CCAPM1 (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
CCAPM1	DBH	name	-	ECOM1	CAPP1	CAPN1	MAT1	TOG1	PWM1	ECCF1

ECOM1 : Comparator Enable bit.
ECOM1=0 disables the comparator function;
ECOM1=1 enables the comparator function.

CAPP1 : Capture Positive control bit.
CAPP1=1 enables positive edge capture.

CAPN1 : Capture Negative control bit.
CAPN1=1 enables negative edge capture.

MAT1 : Match control bit.
When MAT1 = 1, a match of the PCA counter with this module's compare/capture register causes the CCF1 bit in CCON to be set.

TOG1 : Toggle control bit.
When TOG1=1, a match of the PCA counter with this module's compare/capture register causes the CCP1 pin to toggle.
(PCA1/PWM1/P3.5)

PWM1 : Pulse Width Modulation.
PWM1=1 enables the CEX1 pin to be used as a pulse width modulated output.
(PCA1/PWM1/P3.5)

ECCF1 : Enable CCF1 interrupt.
Enables compare/capture flag CCF1 in the CCON register to generate an interrupt.

The operation mode of PCA modules set as shown in the below table.

Setting the operation mode of PCA modules（CCAPMn register，n = 0,1）

-	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn	Function of PCA modules
	0	0	0	0	0	0	0	No operation
	1	0	0	0	0	1	0	8-bit PWM, no interrupt
	1	1	0	0	0	1	1	8-bit PWM output, interrupt can be generated on rising edge.
	1	0	1	0	0	1	1	8-bit PWM output, interrupt can be generated on falling edge.
	1	1	1	0	0	1	1	8-bit PWM output, interrupt can be generated on both rising and falling edges.
	X	1	0	0	0	0	X	16-bit Capture Mode, caputre triggered by the rising edge on CCPn/PCAn pin
	X	0	1	0	0	0	X	16-bit Capture Mode, capture triggered by the falling edge on CCPn/PCAn pin
	X	1	1	0	0	0	X	16-bit Capture Mode, capture triggered by the transition on CCPn/PCAn pin
	1	0	0	1	0	0	X	16-bit software timer
	1	0	0	1	1	0	X	16-bit high-speed output

4. PCA 16-bit Counter — low 8-bit CL and high 8-bit CH

The addresses of CL and CH respectively are E9H and F9H, and their reset value both are 00H. CL and CH are used to save the PCA load value.

5. PCA Capture/Compare register — CCAPnL and CCAPnH

When PCA is used to capture/compare, CCAPnL and CCAPnH are used to save the 16-bit capture value in corresponding block. When PCA is operated in PWM mode, CCAPnL and CCAPnH are used to control the duty cycle of PWM output signal. "n=0 or 1" respectively stand for module 0 and 1. Reset value of regsiters CCAPnL and CCAPnH are both 00H. Their addresses respectively are:

CCAP0L — EAH, CCAP0H — FAH : Capture / Compare register of module 0

CCAP1L — EBH, CCAP1H — FBH : Capture / Compare register of module 1

6. PWM registers of PCA modules : PCA_PWM0 and PCA_PWM1

PCA_PWM0 : PWM register of PCA module 0

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCA_PWM0	F2H	name	-	-	-	-	-	-	EPC0H	EPC0L

B7 ~ B2 : Reserved
EPC0H : Associated with CCAP0H, it is used in PCA PWM mode.
EPC0L : Associated with CCAP0L, it is used in PCA PWM mode.

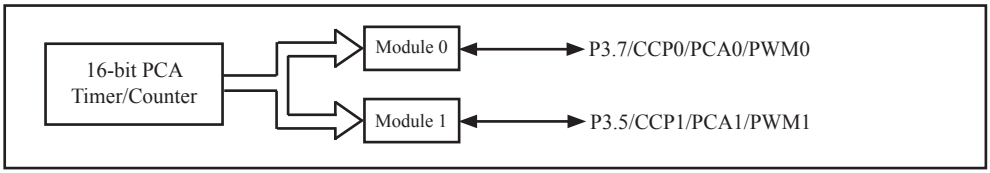
PCA_PWM1 : PWM register of PCA module 1

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
PCA_PWM1	F3H	name	-	-	-	-	-	-	EPC1H	EPC1L

B7 ~ B2 : Reserved
EPC1H : Associated with CCAP1H, it is used in PCA PWM mode.
EPC1L : Associated with CCAP1L, it is used in PCA PWM mode.

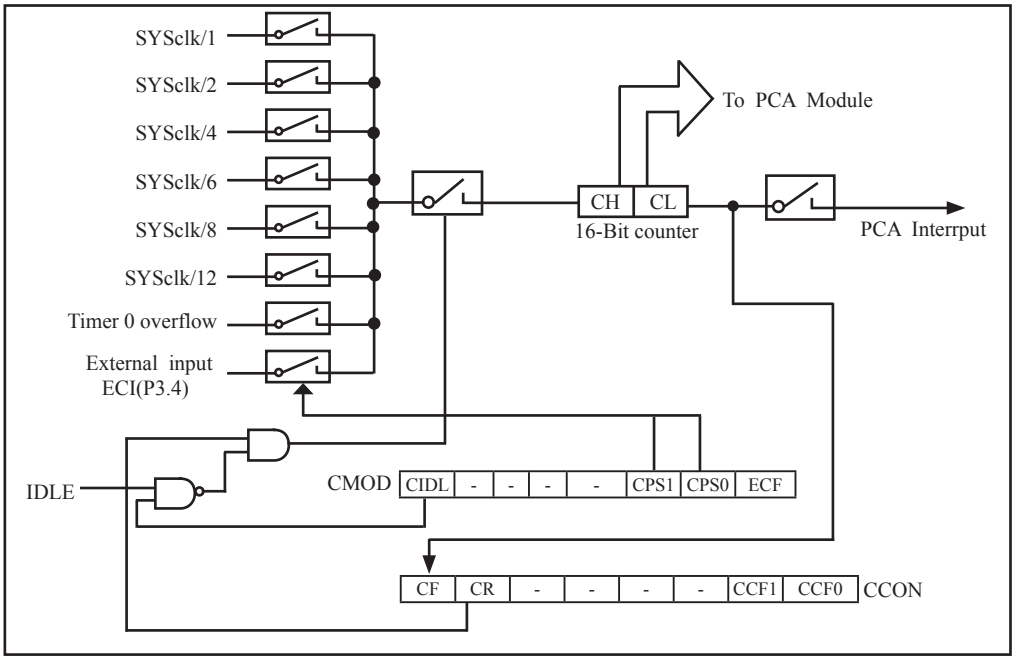
10.2 PCA/PWM Structure

There are 2 channels Programmable Counter Arrary PCA/PWM in STC12C2052AD series MCU.



Programmable Counter Arrary Structure

Each PCA/PWM module can be operated in 4 modes : rising / falling capture mode, software timer, high-speed output mode and adjustable pulse output mode.
STC12C2052AD series: module 0 connect to P3.7/PCA0/PWM0,
module 1 connect to P3.5/PCA1/PWM1.



PCA Timer/Counter

In the CMOD SFR, there are two additional bits associated with the PCA. They are CIDL which allows the PCA to stop during idle mode, and ECF which when set causes an interrupt and the PCA overflow flag CF(in the CCON SFR) to be set when the PCA timer overflows.

The CCON SFR contains the run control bit for PCA and the flags for the PCA timer and each module. To run the PCA the CR bit(CCON.6) must be set by software; oppositely clearing bit CR will shut off PCA is shut off PCA. The CF bit(CCON.7) is set when the PCA counter overflows and an interrupt will be generated if the ECF (CMOD.0) bit in the CMOD register is set. The CF bit can only be cleared by software. There are two bits named CCF0 and CCF1 in SFR CCON. The CCF0 and CCF1 are the flags for module 0 and module 1 respectively. They are set by hardware when either a match or a capture occurs. These flags also can only be cleared by software.

Each module in the PCA has a special function register associated with it, CCAPM0 for module-0 and CCAPM1 for module-1. The register contains the bits that control the mode in which each module will operate. The ECCFn bit controls if to pass the interrupt from CCFn flag in the CCON SFR to the MCU when a match or compare occurs in the associated module. PWMn enables the pulse width modulation mode. The TOGn bit when set causes the pin CCPn output associated with the module to toggle when there is a match between the PCA counter and the module's Capture/Compare register. The match bit (MATn) when set will cause the CCFn bit in the CCON register to be set when there is a match between the PCA counter and the module's Capture/Compare register.

The next two bits CAPNn and CAPPn determine the edge type that a capture input will be active on. The CAPNn bit enables the negative edge, and the CAPPn bit enables the positive edge. If both bits are set, both edges will be enabled and a capture will occur for either transition. The bit ECOMn when set enables the comparator function.

10.3 PCA Modules Operation Mode

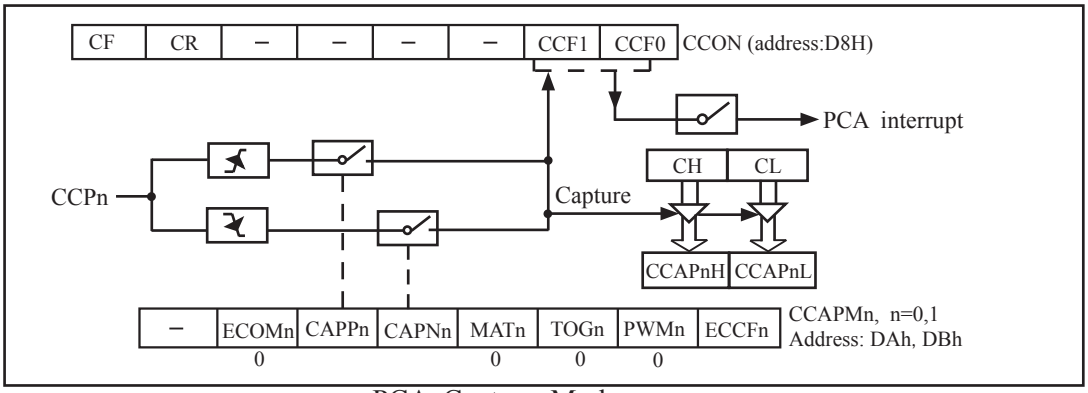
The operation mode of PCA modules set as shown in the below table.

Setting the operation mode of PCA modules（CCAPMn register，n = 0,1）

-	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn	Function of PCA modules
	0	0	0	0	0	0	0	No operation
	1	0	0	0	0	1	0	8-bit PWM, no interrupt
	1	1	0	0	0	1	1	8-bit PWM output, interrupt can be generated on rising edge.
	1	0	1	0	0	1	1	8-bit PWM output, interrupt can be generated on falling edge.
	1	1	1	0	0	1	1	8-bit PWM output, interrupt can be generated on both rising and falling edges.
	X	1	0	0	0	0	X	16-bit Capture Mode, caputre triggered by the rising edge on CCPn/PCAn pin
	X	0	1	0	0	0	X	16-bit Capture Mode, capture triggered by the falling edge on CCPn/PCAn pin
	X	1	1	0	0	0	X	16-bit Capture Mode, capture triggered by the transition on CCPn/PCAn pin
	1	0	0	1	0	0	X	16-bit software timer
	1	0	0	1	1	0	X	16-bit high-speed output

10.3.1 PCA Capture Mode

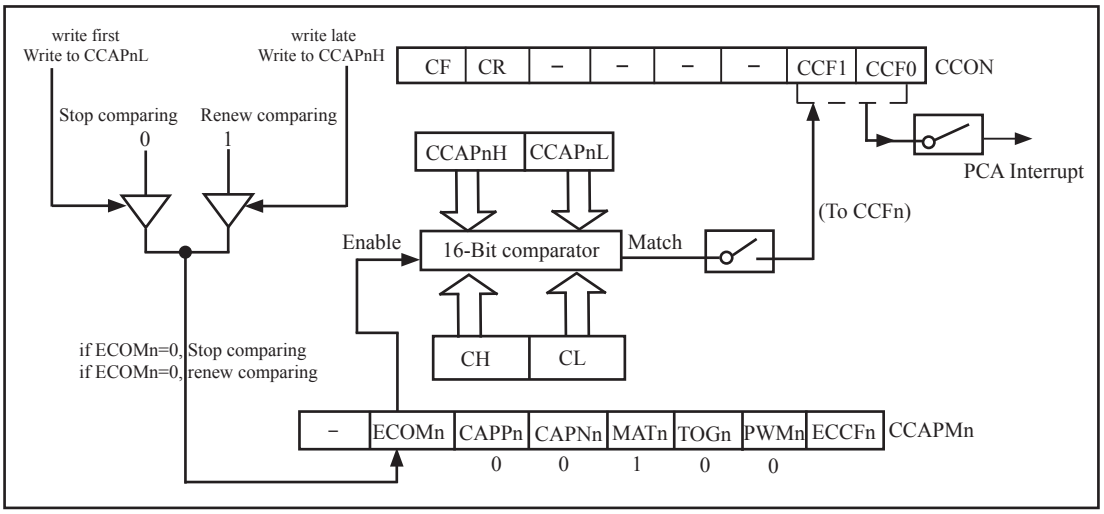
To use one of the PCA modules in the capture mode either one or both of the CCAPM bits – CAPPn and CAPNn, for the module must be set. The external CCPn input (CCP0/P3.7, CCP1/P3.5) for the module is sampled for a transition. When a valid transition occurs, the PCA hardware loads the value of the PCA counter register(CH and CL) into the module's capture registers(CCAPnH and CCAPnL). If the CCFn bit for the module in the CCON SFR and the ECCFn bit in the CCAPMn SFR are set then an interrupt will be generated.



PCA Capture Mode

10.3.2 16-bit Software Timer Mode

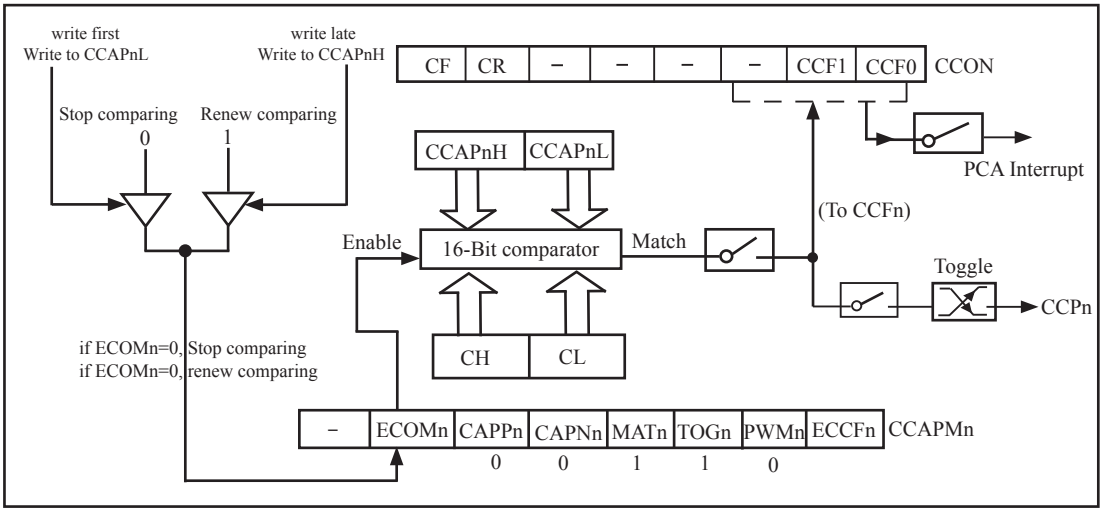
The PCA modules can be used as software timers by setting both the ECOMn and MATn bits in the modules CCAPMn register. The PCA timer will be compared to the module's capture registers and when a match occurs an interrupt will be generated if the CCFn and ECCFn bits for the module are both set.



PCA Software Timer Mode

10.3.3 High Speed Output Mode

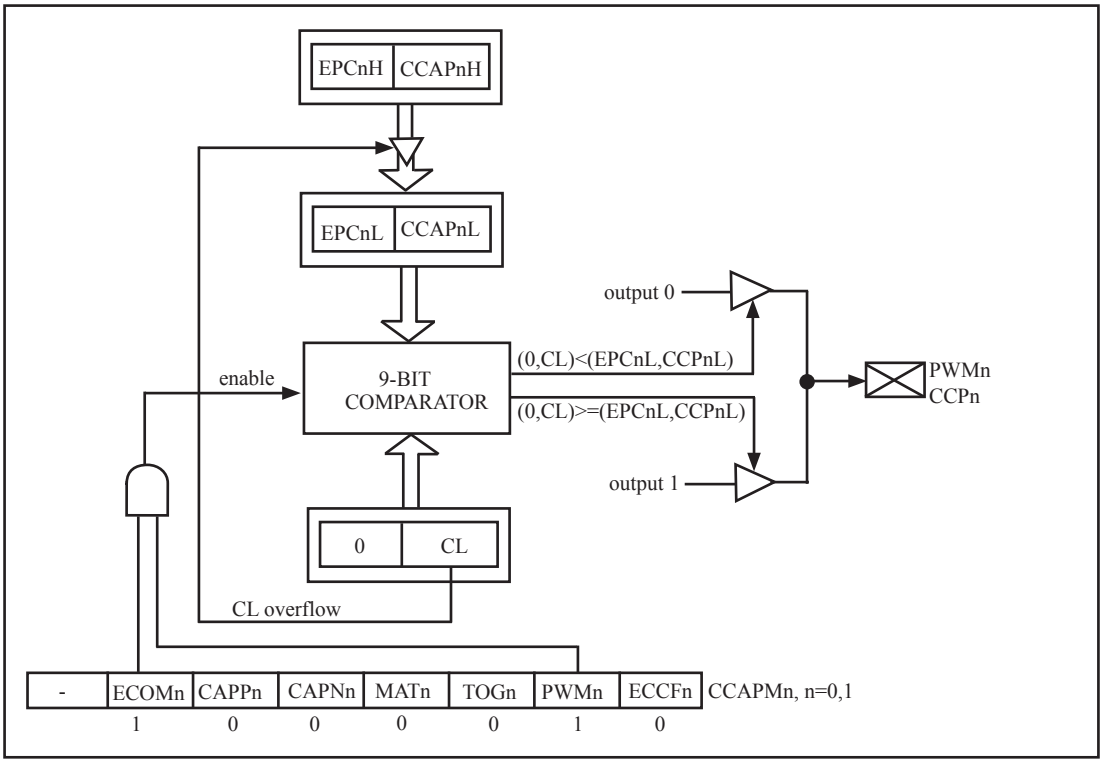
In this mode the CCPn output (port latch) associated with the PCA module will toggle each time a match occurs between the PCA counter and the module's capture registers. To activate this mode the TOGn,MATn,and ECOMn bits in the module's CCAPMn SFR must be set.



PCA High-Speed Output Mode

10.3.4 Pulse Width Modulator Mode (PWM mode)

All of the PCA modules can be used as PWM outputs. The frequency of the output depends on the source for the PCA timer. All of the modules will have the same frequency of output because they all share the same PCA timer. The duty cycle of each module is independently variable using the module' s capture register CCAPnL and EPCnL bit in PCAPWMn register. When the value of the PCA CL SFR is less than the value in the module's {EPCnL,CCAPnL} SFR, the output will be low. When it is equal to or greater than , the output will be high. When CL overflows from FFH to 00H, {EPCnL,CCAPnL} is reloaded with the value in {EPCnH,CCAPnH}. That allows updating the PWM without glitches. The PWMn and ECOMn bits in the module's CCAPMn register must be set to enable the PWM mode.



PCA PWM mode

10.4 Programs for PCA module extended external interrupt (C and ASM)

There are two programs for PCA module extended external interrupt demo, one wrriten in C language and the other in assembly language.

1. C code listing:

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series MCU PCA module Extended external interrupt ----*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------------------*/

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

typedef unsigned char BYTE;
typedef unsigned int WORD;

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

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

void PCA_isr() interrupt 7 using 1
{
        CCF0 = 0;                       //Clear interrupt flag
        PCA_LED = !PCA_LED;             //toggle the test pin while CEX0(P1.3) have a falling edge
}

void main()
{
        CCON = 0;               //Initial PCA control register
                                //PCA timer stop running
                                //Clear CF flag
                                //Clear all module interrupt flag
        CL = 0;                 //Reset PCA base timer
        CH = 0;
        CMOD = 0x00;            //Set PCA timer clock source as Fosc/12
                                //Disable PCA timer overflow interrupt
        CCAPM0 = 0x11;          //PCA module-0 capture by a negative tigger on CEX0(P1.3)
                                //and enable PCA interrupt
//      CCAPM0 = 0x21;          //PCA module-0 capture by a rising edge on CEX0(P1.3)
                                //and enable PCA interrupt
//      CCAPM0 = 0x31;          //PCA module-0 capture by a transition (falling/rising edge)
                                //on CEX0(P1.3) and enable PCA interrupt

        CR = 1;                 //PCA timer start run
        EPCAI = 1;
        EA = 1;

        while (1);
}
```

2. **Assembly code listing:**

```
;/*---------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------------*/
;/* --- STC 1T Series MCU PCA module Extended external interrupt ----------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------------------*/

;/*Declare SFR associated with the PCA */
EPCAI           BIT     IE.6

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

PCA_LED         BIT     P1.0            ;PCA test LED

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

```
        ORG     0033H
PCA_ISR:
        CLR     CCF0                ;Clear interrupt flag
        CPL     PCA_LED             ;toggle the test pin while CEX0(P1.3) have a falling edge
        RETI

;---------------------------------------
        ORG     0100H
MAIN:
        MOV     CCON,   #0          ;Initial PCA control register
                                    ;PCA timer stop running
                                    ;Clear CF flag
                                    ;Clear all module interrupt flag
        CLR     A                   ;
        MOV     CL,     A           ;Reset PCA base timer
        MOV     CH,     A           ;
        MOV     CMOD,   #00H        ;Set PCA timer clock source as Fosc/12
                                    ;Disable PCA timer overflow interrupt
        MOV     CCAPM0,#11H         ;PCA module-0 capture by a falling edge on CEX0(P1.3)
                                    ;and enable PCA interrupt
;       MOV     CCAPM0,#21H         ;PCA module-0 capture by a rising edge on CEX0(P1.3)
                                    ;and enable PCA interrupt
;       MOV     CCAPM0,#31H         ;PCA module-0 capture by a transition (falling/rising edge)
                                    ;on CEX0(P1.3) and enable PCA interrupt
;--------------------------------
        SETB    CR                  ;PCA timer start run
        SETB    EPCAI
        SETB    EA

        SJMP    $
;---------------------------------------
        END
```

10.5 Demo Programs for PCA module acted as 16-bit Timer (C and ASM)

There are two programs for PCA module acted as 16-bit Timer demo, one wrriten in C language and the other in assembly language.

1. C code listing:

```
;/*---------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------------------*/
;/* --- STC 1T Series MCU PCA module acted as 16-bit Timer Demo ---------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------------------*/

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

#define   FOSC     18432000L
#define   T100Hz   (FOSC / 12 / 100)

typedef   unsigned char       BYTE;
typedef   unsigned int        WORD;

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

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

BYTE    cnt;
WORD    value;

void PCA_isr() interrupt 7 using 1
{
        CCF0 = 0;                                       //Clear interrupt flag
        CCAP0L = value;
        CCAP0H = value >> 8;                            //Update compare value
        value += T100Hz;
        if (cnt-- == 0)
        {
                cnt = 100;                              //Count 100 times
                PCA_LED = !PCA_LED;                     //Flash once per second
        }
}

void main()
{
        CCON = 0;                                       //Initial PCA control register
                                                        //PCA timer stop running
                                                        //Clear CF flag
                                                        //Clear all module interrupt flag
        CL = 0;                                         //Reset PCA base timer
        CH = 0;
        CMOD = 0x00;                                    //Set PCA timer clock source as Fosc/12
                                                        //Disable PCA timer overflow interrupt
        value = T100Hz;
        CCAP0L = value;
        CCAP0H = value >> 8;                            //Initial PCA module-0
        value += T100Hz;
        CCAPM0 = 0x49;                                  //PCA module-0 work in 16-bit timer mode
                                                        //and enable PCA interrupt

        CR = 1;                                         //PCA timer start run
        EPCAI = 1;
        EA = 1;
        cnt = 0;

        while (1);
}
```

2. **Assembly code listing:**

```
;/*---------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------------*/
;/* --- STC 1T Series MCU PCA module acted as 16-bit Timer Demo ---------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------------------*/

T100Hz          EQU     3C00H           ;(18432000 / 12 / 100)

;/*Declare SFR associated with the PCA */
EPCAI           BIT     IE.6

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

PCA_LED         BIT     P1.0            ;PCA test LED

CNT             EQU     20H
;----------------------------------------
        ORG     0000H
        LJMP    MAIN
```

```
        ORG     0033H
        LJMP    PCA_ISR
;---------------------------------------
        ORG     0100H
MAIN:
        MOV     SP,     #3FH                    ;Initial stack point
        MOV     CCON,   #0                      ;Initial PCA control register
                                                ;PCA timer stop running
                                                ;Clear CF flag
                                                ;Clear all module interrupt flag
        CLR     A                               ;
        MOV     CL,     A                       ;Reset PCA base timer
        MOV     CH,     A                       ;
        MOV     CMOD,   #00H                    ;Set PCA timer clock source as Fosc/12
                                                ;Disable PCA timer overflow interrupt
;--------------------------------
        MOV     CCAP0L,#LOW T100Hz              ;
        MOV     CCAP0H,#HIGH T100Hz             ;Initial PCA module-0
        MOV     CCAPM0,#49H         ;PCA module-0 work in 16-bit timer mode and enable PCA interrupt
;--------------------------------
        SETB    CR                              ;PCA timer start run
        SETB    EPCAI
        SETB    EA
        MOV     CNT,    #100

        SJMP    $
;----------------------------------------
PCA_ISR:
        PUSH    PSW
        PUSH    ACC
        CLR     CCF0                            ;Clear interrupt flag
        MOV     A,      CCAP0L
        ADD     A,      #LOW T100Hz             ;Update compare value
        MOV     CCAP0L,A
        MOV     A,      CCAP0H
        ADDC    A,      #HIGH T100Hz
        MOV     CCAP0H,A
        DJNZ    CNT,    PCA_ISR_EXIT            ;count 100 times
        MOV     CNT,    #100
        CPL     PCA_LED                         ;Flash once per second
PCA_ISR_EXIT:
        POP     ACC
        POP     PSW
        RETI
;----------------------------------------
        END
```

10.6 Programs for PCA module as 16-bit High Speed Output(C and ASM)

There are two programs for PCA module as 16-bit High Speed Output, one wrriten in C language and the other in assembly language.

1. C code listing:

```
;/*-------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------*/
;/* --- STC 1T Series MCU PCA module as 16-bit High Speed Output ----*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*-------------------------------------------------------------------------------------*/

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

#define   FOSC    18432000L
#define   T100KHz (FOSC / 4 / 100000)

typedef   unsigned char   BYTE;
typedef   unsigned int    WORD;

/*Declare SFR associated with the PCA */
sbit      EPCAI           =  IE^6;

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

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

BYTE    cnt;
WORD    value;

void PCA_isr( ) interrupt 7 using 1
{
        CCF0 = 0;                       //Clear interrupt flag
        CCAP0L = value;
        CCAP0H = value >> 8;            //Update compare value
        value += T100KHz;
}

void main()
{
        CCON = 0;                       //Initial PCA control register
                                        //PCA timer stop running
                                        //Clear CF flag
                                        //Clear all module interrupt flag
        CL = 0;                         //Reset PCA base timer
        CH = 0;
        CMOD = 0x02;                    //Set PCA timer clock source as Fosc/2
                                        //Disable PCA timer overflow interrupt
        value = T100KHz;
        CCAP0L = value;                 //P1.3 output 100KHz square wave
        CCAP0H = value >> 8;            //Initial PCA module-0
        value += T100KHz;
        CCAPM0 = 0x4d;                  //PCA module-0 work in 16-bit timer mode
                                        //and enable PCA interrupt, toggle the output pin CEX0(P1.3)

        CR = 1;                         //PCA timer start run
        EPCAI = 1;
        EA = 1;
        cnt = 0;

        while (1);
}
```

2. **Assembly code listing:**

```
;/*---------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------*/
;/* --- STC 1T Series MCU PCA module as 16-bit High Speed Output ----*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------------------*/

T100KHz         EQU     2EH                     ;(18432000 / 4 / 100000)

;/*Declare SFR associated with the PCA */
CCON            EQU     0D8H                    ;PCA control register
CCF0            BIT     CCON.0                  ;PCA module-0 interrupt flag
CCF1            BIT     CCON.1                  ;PCA module-1 interrupt flag
CR              BIT     CCON.6                  ;PCA timer run control bit
CF              BIT     CCON.7                  ;PCA timer overflow flag
CMOD            EQU     0D9H                    ;PCA mode register
CL              EQU     0E9H                    ;PCA base timer LOW
CH              EQU     0F9H                    ;PCA base timer HIGH
CCAPM0          EQU     0DAH                    ;PCA module-0 mode register
CCAP0L          EQU     0EAH                    ;PCA module-0 capture register LOW
CCAP0H          EQU     0FAH                    ;PCA module-0 capture register HIGH
CCAPM1          EQU     0DBH                    ;PCA module-1 mode register
CCAP1L          EQU     0EBH                    ;PCA module-1 capture register LOW
CCAP1H          EQU     0FBH                    ;PCA module-1 capture register HIGH
CCAPM2          EQU     0DCH                    ;PCA module-2 mode register
CCAP2L          EQU     0ECH                    ;PCA module-2 capture register LOW
CCAP2H          EQU     0FCH                    ;PCA module-2 capture register HIGH
CCAPM3          EQU     0DDH                    ;PCA module-3 mode register
CCAP3L          EQU     0EDH                    ;PCA module-3 capture register LOW
CCAP3H          EQU     0FDH                    ;PCA module-3 capture register HIGH
;------------------------------------------
        ORG     0000H
        LJMP    MAIN

        ORG     0033H
PCA_ISR:
        PUSH    PSW
        PUSH    ACC
        CLR     CCF0                    ;Clear interrupt flag
```

```
        MOV     A,      CCAP0L
        ADD     A,      #T100KHz            ;Update compare value
        MOV     CCAP0L,A
        CLR     A
        ADDC    A,      CCAP0H
        MOV     CCAP0H,A

PCA_ISR_EXIT:
        POP     ACC
        POP     PSW
        RETI
;----------------------------------------
        ORG     0100H
MAIN:
        MOV     CCON,   #0              ;Initial PCA control register
                                        ;PCA timer stop running
                                        ;Clear CF flag
                                        ;Clear all module interrupt flag
        CLR     A                       ;
        MOV     CL,     A               ;Reset PCA base timer
        MOV     CH,     A               ;
        MOV     CMOD,   #02H            ;Set PCA timer clock source as Fosc/2
                                        ;Disable PCA timer overflow interrupt
;--------------------------------
        MOV     CCAP0L,#T100KHz         ;P1.3 output 100KHz square wave
        MOV     CCAP0H,#0               ;Initial PCA module-0
        MOV     CCAPM0,#4dH             ;PCA module-0 work in 16-bit timer mode
                                        ;and enable PCA interrupt, toggle the output pin CEX0(P1.3)
;--------------------------------
        SETB    CR                      ;PCA timer start run
        SETB    EA

        SJMP    $
;----------------------------------------
        END
```

10.7 Demo Programs for PCA module as PWM Output (C and ASM)

1. C code listing:

```
*-------------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------------*/
;/* --- STC 1T Series MCU PCA module output PWM wave Demo ----------*/
 ;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*-------------------------------------------------------------------------------------------*/

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

#define   FOSC    18432000L

typedef   unsigned char     BYTE;
typedef   unsigned int      WORD;

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

```
void main()
{
	CCON = 0;				//Initial PCA control register
						//PCA timer stop running
						//Clear CF flag
						//Clear all module interrupt flag
	CL = 0;					//Reset PCA base timer
	CH = 0;
	CMOD = 0x02;				//Set PCA timer clock source as Fosc/2
						//Disable PCA timer overflow interrupt
	CCAP0H = CCAP0L = 0x80;			//PWM0 port output 50% duty cycle square wave
	CCAPM0 = 0x42;				//PCA module-0 work in 8-bit PWM mode
						//and no PCA interrupt

	CCAP1H = CCAP1L = 0xff;			//PWM1 port output 0% duty cycle square wave
	PCAPWM1 = 0x03;
	CCAPM1 = 0x42;				//PCA module-1 work in 8-bit PWM mode
						//and no PCA interrupt

	 CR = 1;				//PCA timer start run

	while (1);
}
```

2. Assembly code listing:

```
*-------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------------*/
;/* --- STC 1T Series MCU PCA module output PWM wave Demo ----------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*-------------------------------------------------------------------------------------*/

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

;--------------------------------------------
        ORG     0000H
        LJMP    MAIN
;--------------------------------------------
        ORG     0100H
MAIN:
        MOV     CCON,   #0              ;Initial PCA control register
                                        ;PCA timer stop running
                                        ;Clear CF flag
                                        ;Clear all module interrupt flag
```

```
        CLR     A                               ;
        MOV     CL,     A                       ;Reset PCA base timer
        MOV     CH,     A                       ;
        MOV     CMOD,   #02H                    ;Set PCA timer clock source as Fosc/2
                                                ;Disable PCA timer overflow interrupt
;--------------------------------
        MOV     A,      #080H           ;
        MOV     CCAP0H,A                 ;PWM0 port output 50% duty cycle square wave
        MOV     CCAP0L,A                ;
        MOV     CCAPM0,#42H             ;PCA module-0 work in 8-bit PWM mode and no PCA interrupt
;--------------------------------
        MOV     A,      #0C0H           ;
        MOV     CCAP1H,A                ;PWM1 port output 25% duty cycle square wave
        MOV     CCAP1L,A                ;
        MOV     CCAPM1,#42H             ;PCA module-1 work in 8-bit PWM mode and no PCA interrupt
;--------------------------------
        SETB    CR                      ;PCA timer start run

        SJMP    $
;-----------------------------------------
        END
```

10.8 Demo Program for PCA clock base on Timer 1 overflow rate

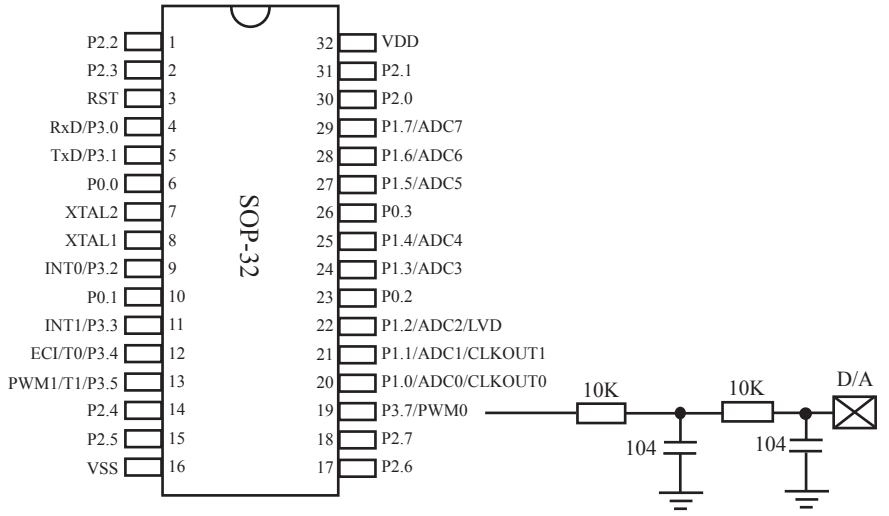
```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------*/
;/* --- STC 1T Series achieve adjustable frequency PWM output-----*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

;/*Declare SFR associated with the ADC */
IPH     EQU 0B7H        ;Interrupt priority control register high byte
CCON    EQU 0D8H        ;PCA control register
CCF0    BIT CCON.0      ;PCA module-0 interrupt flag
CCF1    BIT CCON.1      ;PCA module-1 interrupt flag
CR      BIT CCON.6      ;PCA timer run control bit
CF      BIT CCON.7      ;PCA timer overflow flag
CMOD    EQU 0D9H        ;PCA mode register
CL      EQU 0E9H        ;PCA base timer LOW
CH      EQU 0F9H        ;PCA base timer HIGH
CCAPM0          EQU 0DAH        ;PCA module-0 mode register
CCAP0L EQU 0EAH         ;PCA module-0 capture register LOW
CCAP0H EQU 0FAH         ;PCA module-0 capture register HIGH
CCAPM1          EQU 0DBH        ;PCA module-1 mode register
CCAP1L EQU 0EBH         ;PCA module-1 capture register LOW
CCAP1H EQU 0FBH         ;PCA module-1 capture register HIGH

;/*Define working LED */
LED_MCU_START           EQU P1.7
LED_5ms_Flashing        EQU P1.6
LED_1S_Flashing  EQU P1.5

Channel_5mS_H   EQU 01H         ;PCA module-1 5ms counter high byte @ 18.432MHz
Channel_5mS_L   EQU 00H         ;PCA module-1 5ms counter low byte @ 18.432MHz

Timer0_Reload_1   EQU 0F6H      ;Timer0 auto reload value (-10)
Timer0_Reload_2   EQU 0ECH      ;Timer0 auto reload value (-20)

PWM_WIDTH       EQU 0FFH        ;low duty

Counter   EQU 030H          ;counter value
```

```
;---------------------------------------
        ORG     0000H
        LJMP    MAIN
        ORG     003BH
        LJMP    PCA_interrupt
;---------------------------------------

        ORG     0050H
MAIN:
        CLR     LED_MCU_START           ;Turn on MCU working LED
        MOV     SP,#7FH
        MOV     Counter,#0              ;initial Counter var
        ACALL   PAC_Initial             ;initial PCA
        ACALL   Timer0_Initial          ;Initial Timer0

MAIN_Loop:
;---------------------------------
        MOV     TH0,#Timer0_Reload_1    ;Set Timer0 overload rate 1
        MOV     TL0,#Timer0_Reload_1
        MOV     A,#PWM_WIDTH ;setting duty
        MOV     CCAP0H,A
        ACALL   delay
        MOV     TH0,#Timer0_Reload_2    ;Set Timer0 overload rate 2
        MOV     TL0,#Timer0_Reload_2
        ACALL   delay

;---------------------------------
        MOV     TH0,#Timer0_Reload_1    ;Set Timer0 overload rate 1
        MOV     TL0,#Timer0_Reload_1
        MOV     A,#PWM_WIDTH
        ACALL   RL_A    ;change duty
        ACALL   RL_A
        MOV     CCAP0H,A
        ACALL   delay
        MOV     TH0,#Timer0_Reload_2    ;Set Timer0 overload rate 2
        MOV     TL0,#Timer0_Reload_2
        ACALL   delay

;---------------------------------
        MOV     TH0,#Timer0_Reload_1    ;Set Timer0 overload rate 1
        MOV     TL0,#Timer0_Reload_1
        MOV     A,#PWM_WIDTH
        ACALL   RL_A    ;change duty
```

```
        ACALL   RL_A
        ACALL   RL_A
        ACALL   RL_A
        MOV     CCAP0H,A
        ACALL   delay
        MOV     TH0,#Timer0_Reload_2        ;Set Timer0 overload rate 2
        MOV     TL0,#Timer0_Reload_2
        ACALL   delay

;--------------------------------
        SJMP    MAIN_Loop
;----------------------------------------
RL_A:
        CLR     C
        RRC     A
        RET

;----------------------------------------
Timer0_Initial:
        MOV     TMOD,#02H           ;8-bit auto-reload
        MOV     TH0,#Timer0_Reload_1
        MOV     TL0,#Timer0_Reload_1
        SETB    TR0         ;strat run
        RET

;----------------------------------------
PCA_Initial:
        MOV     CMOD,#10000100B             ;PCA timer base on Timer0
        MOV     CCON,#00H           ;PCA stop count
        MOV     CL,#0       ;initial PCA counter
        MOV     CH,#0
        MOV     CCAPM0,#42H     ;PCA module-0 as 8-bit PWM
        MOV     PCA_PWM0,#0     ;PWM mode 9th bit
;       MOV     PCA_PWM0,#03H ;PWM will keep low level

        MOV     CCAP1L,#Channel_5mS_L   ;initial PCA module-1
        MOV     CCAP1H,#Channel_5mS_H
        MOV     CCAPM1,#49H     ;PCA module-1 as 16-bit timer
        SETB    EA
        SETB    CR          ;PCA counter start running
        RET
```

```
;----------------------------------------
PCA_Interrupt:
        PUSH    ACC
        PUSH    PSW
        CPL     LED_5mS_Flashing            ;Flashing once per 5ms
        MOV     A,#Channel_5mS_L
        ADD     A,CCAP1L
        MOV     CCAP1L,A
        MOV     A,#Channel_5mS_H
        ADDC    A,CCAP1H
        MOV     CCAP1H,A
        CLR     CCF1    ;Clear PCA module-1 flag

        INC     Counter
        MOV     A,Counter
        CLR     C
        SUBB    A,#100  ;Count 100 times
        JC      PCA_Interrupt_Exit
        MOV     Counter,#0
        CPL     LED_1S,Flash
PCA_Interrupt_Exit:
        POP     PSW
        POP     ACC
        RETI
;----------------------------------------
delay:
        CLR     A
        MOV     R1,A
        MOV     R2,A
        MOV     R3,#80H
delay_loop:
        NOP
        NOP
        NOP
        DJNZ    R1,delay_loop
        DJNZ    R2,delay_loop
        DJNZ    R3,delay_loop
        RET
;----------------------------------------
        END
```

10.9 Using PWM achieve D/A Conversion function reference circuit



Chapter 11. Serial Peripheral Interface (SPI)

STC12C2052AD provides another high-speed serial communication interface, the SPI interface. SPI is a full-duplex, high-speed, synchronous communication bus with two operation modes: Master mode and Slave mode. Up to 3Mbit/s can be supported in either Master or Slave mode under the SYSclk=12MHz. Two status flags are provided to signal the transfer completion and write-collision occurrence.

11.1 Special Function Registers related with SPI

SPI Management SFRs

Mnemonic	Description	Address	Bit address and Symbol								Reset Value
			B7	B6	B5	B4	B3	B2	B1	B0	
SPCTL	SPI Control Register	85H	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	0000,0100
SPSTAT	SPI Status Register	84H	SPIF	WCOL	-	-	-	-	-	-	00xx,xxxx
SPDAT	SPI Data Register	86H									0000,0000

1. SPI Control register: SPCTL (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPCTL	85H	name	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SSIG : Control whether SS pin is ignored or not.
If SSIG=1, MSTR(SPCTL.4) decides whether the device is a master or slave.
If SSIG=0, the SS pin decides whether the device is a master or slave. SS pin can be used as I/O port.

SPEN : SPI enable bit.
If SPEN=0, the SPI interface is disabled and all SPI pins will be general-purpose I/O ports.
If SPEN=1, the SPI is enabled.

DORD : Set the transmitted or received SPI data order.
If DORD=1, The LSB of the data word is transmitted first.
If DORD=0, The MSB of the data word is transmitted first.

MSTR : Master/Slave mode select bit.
If MSTR=0, set the SPI to play as Slave part.
If MSTR=1, set the SPI to play as Master part.

CPOL : SPI clock polarity select bit.
If CPOL=1, SPICLK is high level when in idle mode. The leading edge of SPICLK is the falling edge and the trailing edge is the rising edge.
If CPOL=0, SPICLK is low when idle. The leading edge of SPICLK is the rising edge and the trailing edge is the falling edge.

CPHA : SPI clock phase select bit.
If CPHA=1, Data is driven on the leading edge of SPICLK, and is sampled on the trailing edge.
If CPHA=0, Data is driven when SS pin is low (SSIG=0) and changes on the trailing edge of SPICLK. Data is sampled on the leading edge of SPICLK. (Note : If SSIG=1, CPHA must not be 0, otherwise the operation is undefined)

SPR1-SPR0 : SPI clock rate select bit (when in master mode)

SPI clock frequency select bit

SPR1	SPR0	SPI clock (SCLK)
0	0	CPU_CLK/4
0	1	CPU_CLK/16
1	0	CPU_CLK/64
1	1	CPU_CLK/128

CPU_CLK is CPU clock.

When CPHA equals 0, SSIG must be 0 and SS pin must be negated and reasserted between each successive serial byte transfer. If the SPDAT register is written while SS is active(0), a write collision error results and WCOL is set.

When CPHA equals 1, SSIG may be 0 or 1. If SSIG=0, the SS pin may remain active low between successive transfers(can be tied low at any times). This format is sometimes preferred for use in systems having a signle fixed master and a single slave configuration.

2. SPI State register: SPSTAT (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPSTAT	CDH	name	SPIF	WCOL	-	-	-	-	-	-

SPIF : SPI transfer completion flag.
When a serial transfer finishes, the SPIF bit is set and an interrupt is generated if all the EADC_SPI (IE.6) bit, ESPI bit (AUXR.3) and the EA (IE.7) bit are set. If SS is an input and is driven low when SPI is in master mode with SSIG = 0, SPIF will also be set to signal the “mode change”.The SPIF is cleared in software by “writing 1 to this bit”.

WCOL: SPI write collision flag.
The WCOL bit is set if the SPI data register, SPDAT, is written during a data transfer. The WCOL flag is cleared in software by “writing 1 to this bit”.

3. SPI Data register : SPDAT (Non bit-addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPDAT	CFH	name								

The SFR SPDAT holds the data to be transmitted or the data received.

5. Registers related with SPI interrupt : IE, AUXR, IP and IPH

IE: Interrupt Enable Rsgister (Bit-addressable)

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

EA : disables all interrupts.
If EA = 0,no interrupt will be acknowledged.
If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.

EADC_SPI : Interrupt controller of Serial Peripheral Interface (SPI) and A/D Converter (ADC).
If EADC_SPI = 0, Disable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).
If EADC_SPI = 1, Enable the interrupt of Serial Peripheral Interface (SPI) and A/D Converter (ADC).

AUXR: Auxiliary register (Non bit-addressable)

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

ELVDI : Enable/Disable interrupt from low-voltage sensor
0 : (default)Inhibit the low-voltage sensor functional block to generate interrupt to the MCU
1 : Enable the low-voltage sensor functional block to generate interrupt to the MCU

ESPI : Enable/Disable interrupt from Serial Peripheral Interface (SPI)
0 : (default)Inhibit the SPI functional block to generate interrupt to the MCU
1 : Enable the SPI functional block to generate interrupt to the MCU

EADCI : Enable/Disable interrupt from A/D converter
0 : (default) Inhibit the ADC functional block to generate interrupt to the MCU
1 : Enable the ADC functional block to generate interrupt to the MCU

IPH: Interrupt Priority High Register (Non bit-addressable)

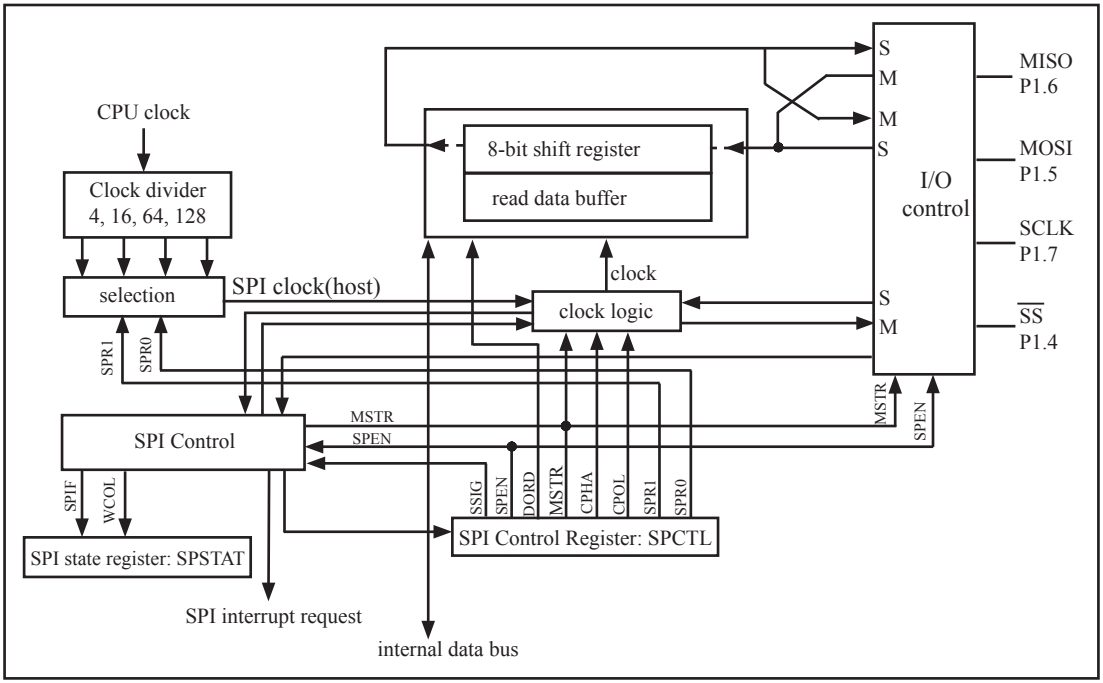
SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
IPH	B7H	name	-	PPCA_LVDH	PADC_SPIH	PSH	PT1H	PX1H	PT0H	PX0H

IP: Interrupt Priority Register (Bit-addressable)

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

PADC_SPIH, PADC_SPI : Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt priority control bits.
if PADC_SPIH=0 and PADC_SPI=0, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 0).
if PADC_SPIH=0 and PADC_SPI=1, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 1).
if PADC_SPIH=1 and PADC_SPI=0, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 2).
if PADC_SPIH=1 and PADC_SPI=1, Serial Peripheral Interface (SPI) and A/D Converter (ADC) interrupt are assigned lowest priority(priority 3).

11.2 SPI Structure



SPI block diagram

The SPI interface has three pins implementing the SPI functionality: SCLK(P1.7), MISO(P1.6), MOSI(P1.5). An extra pin SS(P1.4) is designed to configure the SPI to run under Master or Slave mode. SCLK, MOSI and MISO are typically tied together between two or more SPI devices. Data flows from master to slave on MOSI(Master Out Slave In) pin and flows from slave to master on MISO(Master In Slave Out) pin. The SCLK signal is output in the master mode and is input in the slave mode. If the SPI system is disabled, i.e, SPEN(SPCTL.6)=0, these pins are configured as general-purposed I/O port(P1.4 ~ P1.7).

SS is thel slave select pin. In a typical configuration, an SPI master asserts one of its port pins to select one SPI device as the current slave. An SPI slave device uses its SS pin to determine whether it is selected. But if SPEN=0 or SSIG(SPCTL.7) bit is 1, the SS pin is ignored. Note that even if the SPI is configured as a master(MSTR/SPCTL.4=1), it can still be converted to a slave by driving the SS pin low. When the conversion happened, the SPIF bit(SPSTAT.7) will be set.

Two devices with SPI interface communicate with each other via one synchronous clock signal, one input data signal, and one output data signal. There are two concerns the user should take care, one of them is latching data on the negative edge or positive edge of the clock signal which named polarity, the other is keeping the clock signal low or high while the device idle which named phase. Permuting those states from polarity and phase, there could be four modes formed, they are SPI-MODE-0, SPI-MODE-1, SPI-MODE-2, SPI-MODE-3. Many device declares that they meet SPI machanism, but few of them are adaptive to all four modes. The STC12C5A60S2 series are flexible to be configured to communicate to another device with MODE-0, MODE-1, MODE-2 or MODE-3 SPI, and play part of Master and Slave.

11.3 SPI Data Communication

11.3.1 SPI Configuration

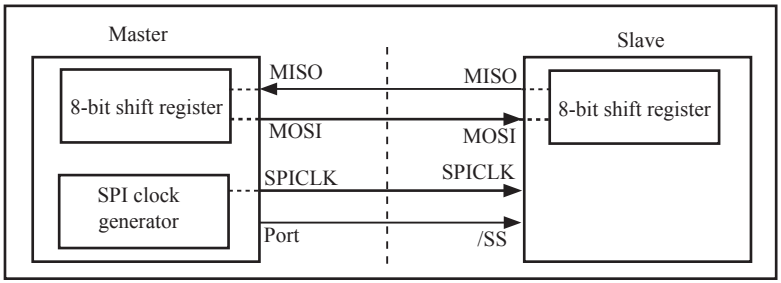
When SPI data communication, SPEN, SSIG, SS(P1.4) and MSTR jointly control the selection of master and slave.

Table: SPI master and slave selection

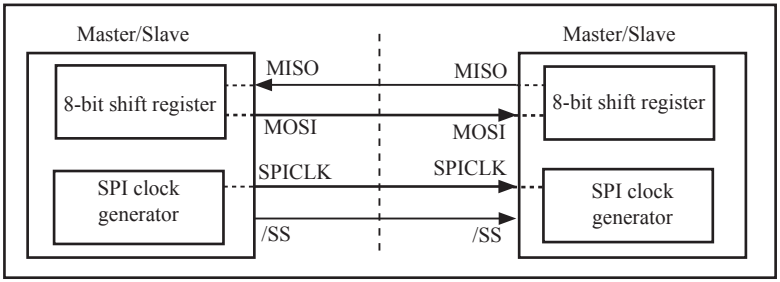
SPEN	SSIG	SS (P1.4)	MSTR	Mode	MISO (P1.6)	MOSI (P1.5)	SPICLK (P1.7)	Remark
0	X	P1.4	X	SPI disable	GPI/O P1.6	GPI/O P1.5	GPI/O P1.7	SPI is disabled. P1.4/P1.5/P1.6/ P1.7 as GPIO.
1	0	0	0	Selected salve	output	input	input	Selected as slave
1	0	1	0	Unselected slave	Hi-Z	input	input	Not selected.
1	0	0	1->0	slave	output	input	input	Convert from Master to Slave
1	0	1	1	Master (idle)	input	high-impedance	high-impedance	MOSI and SCLK are in high-impedance state in order to avoid bus clash when master is idle. Whether is SCLK pulled up or pulled down depends on CPOL/SPCTL.3. But it do not be allowed that SCLK is suspended.
				Master (active)		output	output	MOSI and SCLK is strong push-pull output.
1	1	P1.4	0	slave	output	input	input	Slave
1	1	P1.4	1	Master	input	output	output	Master

11.3.2 SPI Data Communication Modes

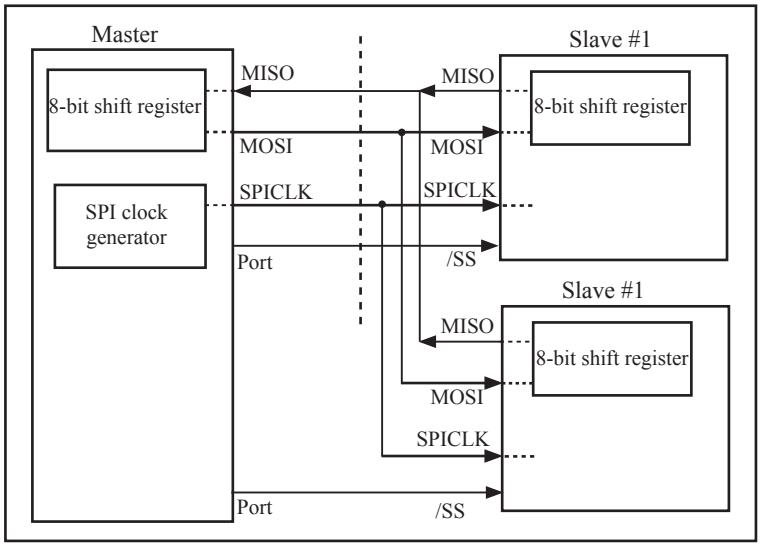
There are 3 SPI data communication modes: single master — single slave, dual devices configuration (both can be a master or slave) and single master — multiple slaves.



SPI single master — single slave configuration



SPI dual device configuration, both can be a master or slave



SPI single master multiple slaves configuration

In SPI, transfers are always initiated by the master. If the SPI is enabled(SPEN=1) and selected as master, any instruction that use SPI data register SPDAT as the destination will starts the SPI clock generator and a data transfer. The data will start to appear on MOSI about one half SPI bit-time to one SPI bit-time after it. Before starting the transfer, the master may select a slave by driving the SS pin of the corresponding device low. Data written to the SPDAT register of the master shifted out of MOSI pin of the master to the MOSI pin of the slave. And at the same time the data in SPDAT register of the selected slave is shifted out of MISO pin to the MISO pin of the master. During one byte transfer, data in the master and in the slave is interchanged. After shifting one byte, the transfer completion flag(SPIF) is set and an interrupt will be created if the SPI interrupt is enabled.

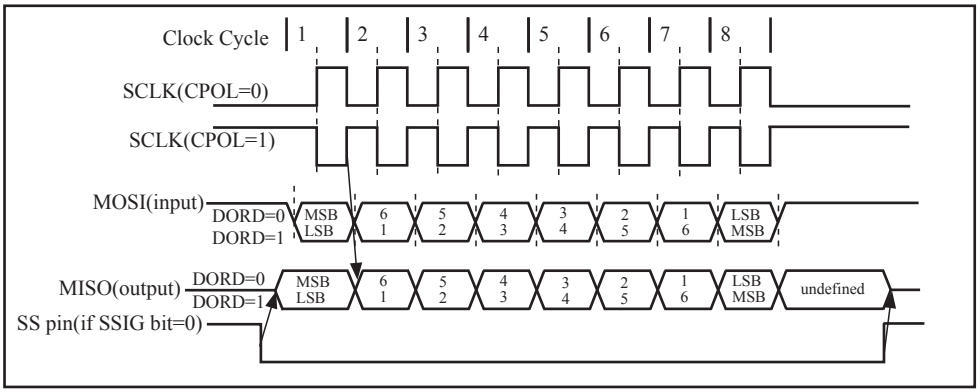
If SPEN=1, SSIG=0, SS pin=1 and MSTR=1, the SPI is enabled in master mode. Before the instruction that use SPDAT as the destination register, the master is in idle state and can be selected as slave device by any other master drives the idle master SS pin low. Once this happened, MSTR bit of the idle master is cleared by hardware and changes its state a selected slave. User software should always check the MSTR bit. If this bit is cleared by the mode change of SS pin and the user wants to continue to use the SPI as a master later, the user must set the MSTR bit again, otherwise it will always stay in slave mode.

The SPI is single buffered in transmit direction and double buffered in receive direction. New data for transmission can not be written to the shift register until the previous transaction is complete. The WCOL bit is set to signal data collision when the data register is written during transaction. In this case, the data currently being transmitted will continue to be transmitted, but the new data which causing the collision will be lost. For receiving data, received data is transferred into a internal parallel read data buffer so that the shift register is free to accept a second byte. However, the received byte must be read from the data register(SPDAT) before the next byte has been completely transferred. Otherwise the previous byte is lost. WCOL can be cleared in software by “writing 1 to the bit”.

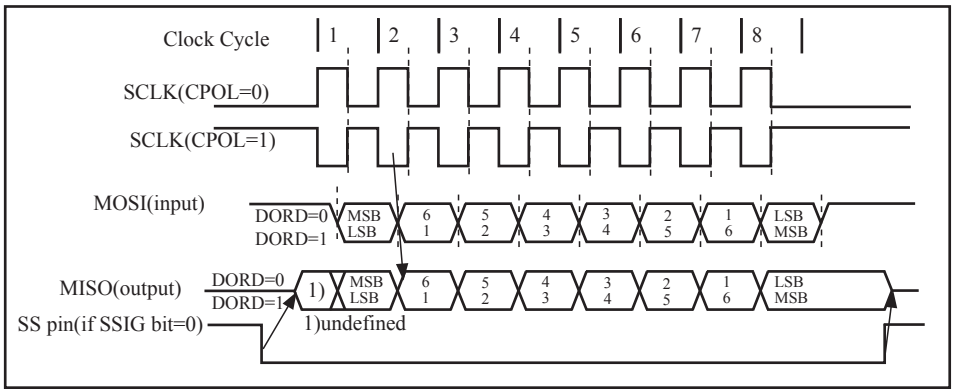
11.3.3 SPI Data Modes

CPHA/SPCTL.2 is SPI clock phase select bit which is used to setting the clock edge of Data sample and change. CPOL/SPCTL.3 is used to select SPI clock polarity.

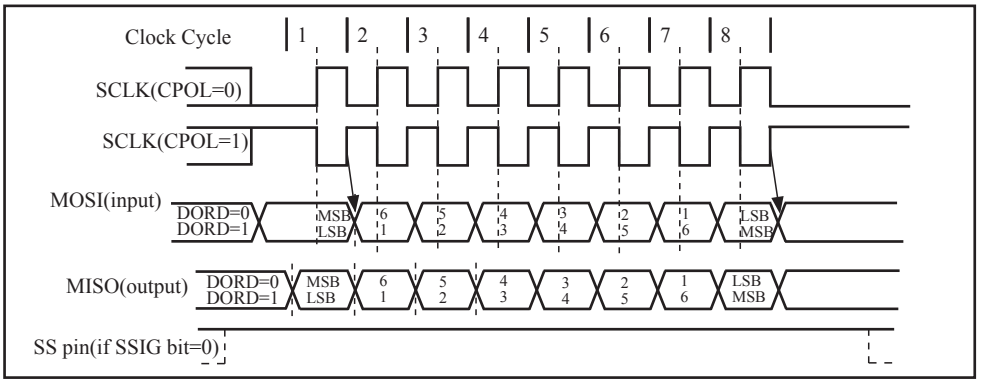
The following are some typical timing diagrams which depend on the value of CPHA/SPCTL.2



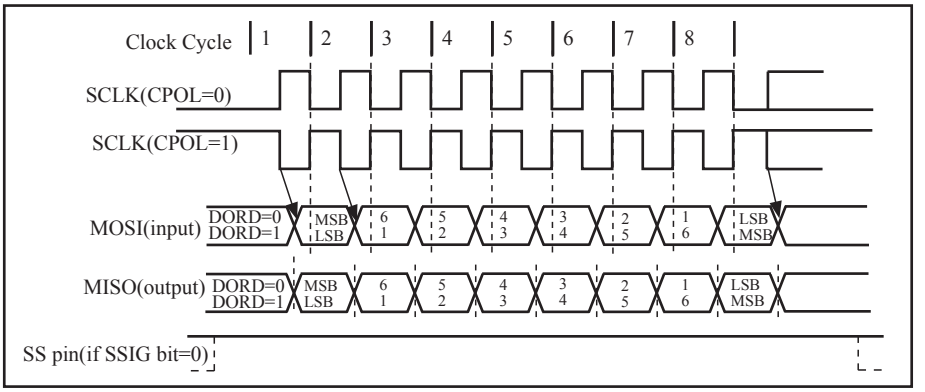
SPI slave transfer format with CPHA=0



SPI slave transfer format with CPHA=1



SPI master transfer format with CPHA=0



SPI master transfer format with CPHA=1

**When P4SPI bit in AUXR1 register is set, the function of SPI is redirected from P3[7:4] to P4[7:4] pin by pin.*

11.4 SPI Function Demo Programs (Single Master — Single Slave)

11.4.1 SPI Function Demo Programs using Interrupts (C and ASM)

The following program,written in C language and assembly language, tests SPI function and applys to SPI single master single slave configuration.

1. C code listing:

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

#include "reg51.h"

#define   MASTER                                  //define:master undefine:slave
#define   FOSC    18432000L
#define   BAUD    (256 - FOSC / 32 / 115200)

typedef   unsigned char       BYTE;
typedef   unsigned int        WORD;
typedef   unsigned long       DWORD;

sfr       AUXR    =   0x8e;                       //Auxiliary register

sfr       SPSTAT  =   0x84;                       //SPI status register
#define   SPIF    0x80                            //SPSTAT.7
#define   WCOL    0x40                            //SPSTAT.6
sfr       SPCTL   =   0x85;                       //SPI control register
#define   SSIG    0x80                            //SPCTL.7
#define   SPEN    0x40                            //SPCTL.6
#define   DORD    0x20                            //SPCTL.5
#define   MSTR    0x10                            //SPCTL.4
#define   CPOL    0x08                            //SPCTL.3
#define   CPHA    0x04                            //SPCTL.2
#define   SPDHH   0x00                            //CPU_CLK/4
#define   SPDH    0x01                            //CPU_CLK/16
#define   SPDL    0x02                            //CPU_CLK/64
#define   SPDLL   0x03                            //CPU_CLK/128
sfr       SPDAT   =   0x86;                       //SPI data register
sbit      SPISS   =   P1^3;                       //SPI slave select, connect to slave' SS(P1.4) pin

sbit      EADC_SPI  =  IE^5;
#define   ESPI    0x08;                           //AUXR.3
```

```
void InitUart();
void InitSPI();
void SendUart(BYTE dat);                    //send data to PC
BYTE RecvUart();                            //receive data from PC

/////////////////////////////////////////////////////////

void main()
{
        InitUart();                         //initial UART
        InitSPI();                          //initial SPI
        AUXR |= ESPI;
        EADC_SPI = 1;
        EA = 1;

        while (1)
        {
        #ifdef  MASTER                      //for master (receive UART data from PC and send it to slave,
                                            //in the meantime receive SPI data from slave and send it to PC)
                ACC = RecvUart();
                SPISS = 0;                  //pull low slave SS
                SPDAT = ACC;                //trigger SPI send
        #endif
        }
}

/////////////////////////////////////////////////////////

void spi_isr()      interrupt 5   using 1   //SPI interrupt routine 5 (002BH)
{
        SPSTAT = SPIF | WCOL;               //clear SPI status
#ifdef MASTER
        SPISS = 1;                          //push high slave SS
        SendUart(SPDAT);                    //return received SPI data
#else                                       //for salve (receive SPI data from master and
        SPDAT = SPDAT;                      //          send previous SPI data to master)
#endif
}

/////////////////////////////////////////////////////////
```

```
void InitUart()
{
	SCON = 0x5a;			//set UART mode as 8-bit variable baudrate
	TMOD = 0x20;			//timer1 as 8-bit auto reload mode
	AUXR = 0x40;			//timer1 work at 1T mode
	TH1 = TL1 = BAUD;		//115200 bps
	TR1 = 1;
}

/////////////////////////////////////////////////////////

void InitSPI()
{
	SPDAT = 0;			//initial SPI data
	SPSTAT = SPIF | WCOL;		//clear SPI status
#ifdef MASTER
	SPCTL = SPEN | MSTR;		//master mode
#else
	SPCTL = SPEN;			//slave mode
#endif
}

/////////////////////////////////////////////////////////

void SendUart(BYTE dat)
{
	while (!TI);			//wait pre-data sent
	TI = 0;				//clear TI flag
	SBUF = dat;			//send current data
}

/////////////////////////////////////////////////////////

BYTE RecvUart()
{
	while (!RI);			//wait receive complete
	RI = 0;				//clear RI flag
	return SBUF;			//return receive data
}
```

2. Assemly code listing:

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

//#define  MASTER                          //define:master undefine:slave

AUXR    DATA    08EH                     ;Auxiliary register
SPSTAT  DATA    084H                     ;SPI status register
SPIF    EQU     080H                     ;SPSTAT.7
WCOL    EQU     040H                     ;SPSTAT.6
SPCTL   DATA    085H                     ;SPI control register
SSIG    EQU     080H                     ;SPCTL.7
SPEN    EQU     040H                     ;SPCTL.6
DORD    EQU     020H                     ;SPCTL.5
MSTR    EQU     010H                     ;SPCTL.4
CPOL    EQU     008H                     ;SPCTL.3
CPHA    EQU     004H                     ;SPCTL.2
SPDHH   EQU     000H                     ;CPU_CLK/4
SPDH    EQU     001H                     ;CPU_CLK/16
SPDL    EQU     002H                     ;CPU_CLK/64
SPDLL   EQU     003H                     ;CPU_CLK/128
SPDAT   DATA    086H                     ;SPI data register
SPISS   BIT     P1.3                     ;SPI slave select, connect to slave' SS(P1.4) pin

EADC_SPI        BIT     IE.5
ESPI            EQU     08H              ;AUXR.3

;///////////////////////////////////////////////////////////

        ORG     0000H
        LJMP    RESET

        ORG     002BH                    ;SPI interrupt routine 5(2BH)
SPI_ISR:
        PUSH    ACC
        PUSH    PSW
        MOV     SPSTAT, #SPIF | WCOL     ;clear SPI status
```

```
#ifdef  MASTER
        SETB    SPISS                   ;push high slave SS
        MOV     A,      SPDAT           ;return received SPI data
        LCALL   SEND_UART
#else                                   //for salve (receive SPI data from master and
        MOV     SPDAT,  SPDAT           ;       send previous SPI data to master)
#endif
        POP     PSW
        POP     ACC
        RETI

;/////////////////////////////////////////////////////////

        ORG     0100H
RESET:
        LCALL   INIT_UART               ;initial UART
        LCALL   INIT_SPI                ;initial SPI
        ORL     AUXR,   #ESPI           ;enable SPI interrupt
        SETB    EADC_SPI
        SETB    EA
MAIN:
#ifdef MASTER                           //for master (receive UART data from PC and send it to slave,
        LCALL   RECV_UART               ; in the meantimereceive SPI data from slave and send it to PC)
        CLR     SPISS                   ;pull low slave SS
        MOV     SPDAT,  A               ;trigger SPI send
#endif
        SJMP    MAIN

;/////////////////////////////////////////////////////////

INIT_UART:
        MOV     SCON,   #5AH            ;set UART mode as 8-bit variable baudrate
        MOV     TMOD,   #20H            ;timer1 as 8-bit auto reload mode
        MOV     AUXR,   #40H            ;timer1 work at 1T mode
        MOV     TL1,    #0FBH           ;115200 bps(256 - 18432000 / 32 / 115200)
        MOV     TH1,    #0FBH
        SETB    TR1
        RET

;/////////////////////////////////////////////////////////
```

```
INIT_SPI:
        MOV     SPDAT,  #0                      ;initial SPI data
        MOV     SPSTAT, #SPIF | WCOL            ;clear SPI status
#ifdef MASTER
        MOV     SPCTL,  #SPEN | MSTR            ;master mode
#else
        MOV     SPCTL,  #SPEN                   ;slave mode
#endif
        RET

;/////////////////////////////////////////////////////////

SEND_UART:
        JNB     TI,     $                       ;wait pre-data sent
        CLR     TI                              ;clear TI flag
        MOV     SBUF,   A                       ;send current data
        RET

;/////////////////////////////////////////////////////////

RECV_UART:
        JNB     RI,$                            ;wait receive complete
        CLR     RI                              ;clear RI flag
        MOV     A,      SBUF                    ;return receive data
        RET
        RET

;/////////////////////////////////////////////////////////

        END
```

11.4.2 SPI Function Demo Programs using Polling (C and ASM)

1. C code listing:

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

#include "reg51.h"

//#define  MASTER                                  //define:master undefine:slave
#define   FOSC              18432000L
#define   BAUD              (256 - FOSC / 32 / 115200)

typedef   unsigned char     BYTE;
typedef   unsigned int      WORD;
typedef   unsigned long     DWORD;

sfr       AUXR   =  0x8e;                         //Auxiliary register

sfr       SPSTAT =  0x84;                         //SPI status register
#define   SPIF    0x80                            //SPSTAT.7
#define   WCOL    0x40                            //SPSTAT.6
sfr       SPCTL  =  0x85;                         //SPI control register
#define   SSIG    0x80                            //SPCTL.7
#define   SPEN    0x40                            //SPCTL.6
#define   DORD    0x20                            //SPCTL.5
#define   MSTR    0x10                            //SPCTL.4
#define   CPOL    0x08                            //SPCTL.3
#define   CPHA    0x04                            //SPCTL.2
#define   SPDHH   0x00                            //CPU_CLK/4
#define   SPDH    0x01                            //CPU_CLK/16
#define   SPDL    0x02                            //CPU_CLK/64
#define   SPDLL   0x03                            //CPU_CLK/128
sfr       SPDAT  =  0x86;                         //SPI data register
sbit      SPISS  =  P1^3;                         //SPI slave select, connect to slave' SS(P1.4) pin

void      InitUart();
void      InitSPI();
void      SendUart(BYTE dat);                     //send data to PC
BYTE      RecvUart();                             //receive data from PC
BYTE      SPISwap(BYTE dat);                      //swap SPI data between master and slave
```

```
//////////////////////////////////////////////////////////

void main()
{
	InitUart();				//initial UART
	InitSPI();				//initial SPI

	while (1)
	{
	#ifdef MASTER		//for master (receive UART data from PC and send it to slave,
				//  in the meantime receive SPI data from slave and send it to PC)
		SendUart(SPISwap(RecvUart()));
	#else				//for salve (receive SPI data from master and
		ACC = SPISwap(ACC);	//         send previous SPI data to master)
	#endif
	}
}

//////////////////////////////////////////////////////////

void InitUart()
{
	SCON = 0x5a;				//set UART mode as 8-bit variable baudrate
	TMOD = 0x20;				//timer1 as 8-bit auto reload mode
	AUXR = 0x40;				//timer1 work at 1T mode
	TH1 = TL1 = BAUD;			//115200 bps
	TR1 = 1;
}

//////////////////////////////////////////////////////////

void InitSPI()
{
	SPDAT = 0;				//initial SPI data
	SPSTAT = SPIF | WCOL;			//clear SPI status
#ifdef MASTER
	SPCTL = SPEN | MSTR;			//master mode
#else
	SPCTL = SPEN;				//slave mode
#endif
}
```

```
//////////////////////////////////////////////////////////

void SendUart(BYTE dat)
{
        while (!TI);                    //wait pre-data sent
        TI = 0;                         //clear TI flag
        SBUF = dat;                     //send current data
}

//////////////////////////////////////////////////////////

BYTE RecvUart()
{
        while (!RI);                    //wait receive complete
        RI = 0;                         //clear RI flag
        return SBUF;                    //return receive data
}

//////////////////////////////////////////////////////////

BYTE SPISwap(BYTE dat)
{
#ifdef MASTER
        SPISS = 0;                      //pull low slave SS
#endif
        SPDAT = dat;                    //trigger SPI send
        while (!(SPSTAT & SPIF));       //wait send complete
        SPSTAT = SPIF | WCOL;           //clear SPI status
#ifdef MASTER
        SPISS = 1;                      //push high slave SS
#endif
        return SPDAT;                   //return received SPI data
}
```

2. Assemly code listing:

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

//#define  MASTER                          //define:master undefine:slave

AUXR    DATA    08EH                    ;Auxiliary register
SPSTAT  DATA    084H                    ;SPI status register
SPIF    EQU     080H                    ;SPSTAT.7
WCOL    EQU     040H                    ;SPSTAT.6
SPCTL   DATA    085H                    ;SPI control register
SSIG    EQU     080H                    ;SPCTL.7
SPEN    EQU     040H                    ;SPCTL.6
DORD    EQU     020H                    ;SPCTL.5
MSTR    EQU     010H                    ;SPCTL.4
CPOL    EQU     008H                    ;SPCTL.3
CPHA    EQU     004H                    ;SPCTL.2
SPDHH   EQU     000H                    ;CPU_CLK/4
SPDH    EQU     001H                    ;CPU_CLK/16
SPDL    EQU     002H                    ;CPU_CLK/64
SPDLL   EQU     003H                    ;CPU_CLK/128
SPDAT   DATA    086H                    ;SPI data register
SPISS   BIT     P1.3                    ;SPI slave select, connect to slave' SS(P1.4) pin

;//////////////////////////////////////////////////////////

        ORG     0000H
        LJMP    RESET
        ORG     0100H
RESET:
        LCALL   INIT_UART               ;initial UART
        LCALL   INIT_SPI                ;initial SPI
```

```
MAIN:
#ifdef  MASTE                   //for master (receive UART data from PC and send it to slave, in the meantime
        LCALL   RECV_UART               ;       receive SPI data from slave and send it to PC)
        LCALL   SPI_SWAP
        LCALL   SEND_UART
#else                                   //for salve (receive SPI data from master and
        LCALL   SPI_SWAP                ;       send previous SPI data to master)
#endif
        SJMP    MAIN
;/////////////////////////////////////////////////////////

INIT_UART:
        MOV     SCON,   #5AH            ;set UART mode as 8-bit variable baudrate
        MOV     TMOD,   #20H            ;timer1 as 8-bit auto reload mode
        MOV     AUXR,   #40H            ;timer1 work at 1T mode
        MOV     TL1,    #0FBH           ;115200 bps(256 - 18432000 / 32 / 115200)
        MOV     TH1,    #0FBH
        SETB    TR1
        RET

;/////////////////////////////////////////////////////////

INIT_SPI:
        MOV     SPDAT,  #0              ;initial SPI data
        MOV     SPSTAT, #SPIF | WCOL    ;clear SPI status
#ifdef  MASTER
        MOV     SPCTL,  #SPEN | MSTR    ;master mode
#else
        MOV     SPCTL,  #SPEN           ;slave mode
#endif
        RET

;/////////////////////////////////////////////////////////

SEND_UART:
        JNB     TI,     $               ;wait pre-data sent
        CLR     TI                      ;clear TI flag
        MOV     SBUF,   A               ;send current data
        RET

;/////////////////////////////////////////////////////////
```

```
RECV_UART:
        JNB     RI,     $                   ;wait receive complete
        CLR     RI                          ;clear RI flag
        MOV     A,      SBUF                ;return receive data
        RET
        RET

;//////////////////////////////////////////////////////////

SPI_SWAP:
#ifdef MASTER
        CLR     SPISS                       ;pull low slave SS
#endif
        MOV     SPDAT,  A                   ;trigger SPI send
WAIT:
        MOV     A,      SPSTAT
        JNB     ACC.7,  WAIT                ;wait send complete
        MOV     SPSTAT, #SPIF | WCOL        ;clear SPI status
#ifdef MASTER
        SETB    SPISS                       ;push high slave SS
#endif
        MOV     A,      SPDAT               ;return received SPI data
        RET

;//////////////////////////////////////////////////////////

        END
```

11.5 SPI Function Demo Programs (Each other as the Master-Slave)

11.5.1 SPI Function Demo Programs using Interrupts (C and ASM)

1. C code listing:

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------*/
/* --- STC12C5Axx Series MCU SPI Demo(Each other as the master-slave) --*/
/* If you want to use the program or the program referenced in the  ----------*/
/* article, please specify in which data and procedures from STC    ----------*/
/*---------------------------------------------------------------------------------------*/

#include "reg51.h"

#define FOSC        18432000L
#define BAUD        (256 - FOSC / 32 / 115200)

typedef   unsigned char       BYTE;
typedef   unsigned int        WORD;
typedef   unsigned long       DWORD;

sfr       AUXR    =   0x8e;                         //Auxiliary register

sfr       SPSTAT  =   0x84;                         //SPI status register
#define   SPIF        0x80                          //SPSTAT.7
#define   WCOL        0x40                          //SPSTAT.6
sfr       SPCTL   =   0x85;                         //SPI control register
#define   SSIG    0x80                              //SPCTL.7
#define   SPEN    0x40                              //SPCTL.6
#define   DORD    0x20                              //SPCTL.5
#define   MSTR    0x10                              //SPCTL.4
#define   CPOL    0x08                              //SPCTL.3
#define   CPHA    0x04                              //SPCTL.2
#define   SPDHH   0x00                              //CPU_CLK/4
#define   SPDH    0x01                              //CPU_CLK/16
#define   SPDL    0x02                              //CPU_CLK/64
#define   SPDLL   0x03                              //CPU_CLK/128
sfr       SPDAT   =         0x86;                                 //SPI data register
sbit      SPISS   =         P1^3;           //SPI slave select, connect to other MCU's SS(P1.4) pin

sbit      EADC_SPI  =  IE^5;
#define   ESPI      0x08;                   //AUXR.3
```

```
void InitUart();
void InitSPI();
void SendUart(BYTE dat);                    //send data to PC
BYTE RecvUart();                            //receive data from PC

bit MSSEL;                                  //1: master 0:slave

/////////////////////////////////////////////////////////

void main()
{
        InitUart();             //initial UART
        InitSPI();              //initial SPI
        AUXR |= ESPI;
        EADC_SPI = 1;
        EA = 1;

        while (1)
        {
                if (RI)
                {
                        SPCTL = SPEN | MSTR;            //set as master
                        MSSEL = 1;
                        ACC = RecvUart();
                        SPISS = 0;                      //pull low slave SS
                        SPDAT = ACC;                    //trigger SPI send
                }
        }
}

/////////////////////////////////////////////////////////

void spi_isr( ) interrupt 5 using 1                     //SPI interrupt routine 5 (002BH)
{
        SPSTAT = SPIF | WCOL;                           //clear SPI status
        if (MSSEL)
        {
                SPCTL = SPEN;                           //reset as slave
                MSSEL = 0;
                SPISS = 1;                              //push high slave SS
                SendUart(SPDAT);                        //return received SPI data
        }
        else
        {                                               //for salve (receive SPI data from master and
                SPDAT = SPDAT;                          //         send previous SPI data to master)
   }
}
/////////////////////////////////////////////////////////
```

```
void InitUart()
{
	SCON = 0x5a;			//set UART mode as 8-bit variable baudrate
	TMOD = 0x20;			//timer1 as 8-bit auto reload mode
	AUXR = 0x40;			//timer1 work at 1T mode
	TH1 = TL1 = BAUD;		//115200 bps
	TR1 = 1;
}

/////////////////////////////////////////////////////////

void InitSPI()
{
	SPDAT = 0;			//initial SPI data
	SPSTAT = SPIF | WCOL;		//clear SPI status
	SPCTL = SPEN;			//slave mode
}

/////////////////////////////////////////////////////////

void SendUart(BYTE dat)
{
	while (!TI);			//wait pre-data sent
	TI = 0;				//clear TI flag
	SBUF = dat;			//send current data
}

/////////////////////////////////////////////////////////

BYTE RecvUart()
{
	while (!RI);			//wait receive complete
	RI = 0;				//clear RI flag
	return SBUF;			//return receive data
}
```

2**. Assembly code listing:**

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------*/
/* --- STC12C5Axx Series MCU SPI Demo(Each other as the master-slave) --*/
/* If you want to use the program or the program referenced in the  ----------*/
/* article, please specify in which data and procedures from STC    ----------*/
/*---------------------------------------------------------------------------------------*/

AUXR    DATA    08EH                    ;Auxiliary register
SPSTAT  DATA    084H                    ;SPI status register
SPIF    EQU     080H                    ;SPSTAT.7
WCOL    EQU     040H                    ;SPSTAT.6
SPCTL   DATA    085H                    ;SPI control register
SSIG    EQU     080H                    ;SPCTL.7
SPEN    EQU     040H                    ;SPCTL.6
DORD    EQU     020H                    ;SPCTL.5
MSTR    EQU     010H                    ;SPCTL.4
CPOL    EQU     008H                    ;SPCTL.3
CPHA    EQU     004H                    ;SPCTL.2
SPDHH   EQU     000H                    ;CPU_CLK/4
SPDH    EQU     001H                    ;CPU_CLK/16
SPDL    EQU     002H                    ;CPU_CLK/64
SPDLL   EQU     003H                    ;CPU_CLK/128
SPDAT   DATA    086H                    ;SPI data register
SPISS   BIT     P1.3                    ;SPI slave select, connect to other MCU's SS(P1.4) pin

EADC_SPI        BIT     IE.5
ESPI            EQU     08H             ;AUXR.3

MSSEL   BIT     20H.0                   ;1: master 0:slave

;///////////////////////////////////////////////////////

        ORG     0000H
        LJMP    RESET

        ORG     002BH                   ;SPI interrupt routine
SPI_ISR:
        PUSH    ACC
        PUSH    PSW
        MOV     SPSTAT, #SPIF | WCOL    ;clear SPI status
        JBC     MSSEL,  MASTER_SEND
```

```
SLAVE_RECV:
                                                    //for salve (receive SPI data from master and
        MOV     SPDAT,  SPDAT                       ;       send previous SPI data to master)
        JMP     SPI_EXIT
MASTER_SEND:
        SETB    SPISS                               ;push high slave SS
        MOV     SPCTL,  #SPEN                       ;           ;reset as slave
        MOV     A,      SPDAT                       ;return received SPI data
        LCALL   SEND_UART
SPI_EXIT:
        POP     PSW
        POP     ACC
        RETI

;/////////////////////////////////////////////////////////

        ORG     0100H
RESET:
        MOV     SP,#3FH
        LCALL   INIT_UART                           ;initial UART
        LCALL   INIT_SPI                            ;initial SPI
        ORL     AUXR,   #ESPI
        SETB    EADC_SPI
        SETB    EA
MAIN:
        JNB     RI,     $                           ;wait UART data
        MOV     SPCTL,  #SPEN | MSTR                ; ;set as master
        SETB    MSSEL
        LCALL   RECV_UART                           ;receive UART data from PC
        CLR     SPISS                               ;pull low slave SS
        MOV     SPDAT, A                                    ;trigger SPI send
        SJMP    MAIN

;/////////////////////////////////////////////////////////

INIT_UART:
        MOV     SCON,   #5AH                        ;set UART mode as 8-bit variable baudrate
        MOV     TMOD,   #20H                        ;timer1 as 8-bit auto reload mode
        MOV     AUXR    ,#40H                       ;timer1 work at 1T mode
        MOV     TL1,    #0FBH                       ;115200 bps(256 - 18432000 / 32 / 115200)
        MOV     TH1,    #0FBH
        SETB    TR1
        RET
```

```
;/////////////////////////////////////////////////////////

INIT_SPI:
        MOV     SPDAT,  #0                  ;initial SPI data
        MOV     SPSTAT, #SPIF | WCOL        ;clear SPI status
        MOV     SPCTL,  #SPEN               ;slave mode
        RET

;/////////////////////////////////////////////////////////

SEND_UART:
        JNB     TI,     $                   ;wait pre-data sent
        CLR     TI                          ;clear TI flag
        MOV     SBUF,   A                   ;send current data
        RET

;/////////////////////////////////////////////////////////

RECV_UART:
        JNB     RI,     $                   ;wait receive complete
        CLR     RI                          ;clear RI flag
        MOV     A,      SBUF                ;return receive data
        RET
        RET

;/////////////////////////////////////////////////////////

        END
```

11.5.2 SPI Function Demo Programs using Polling

1. C code listing:

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------*/
/* --- STC12C5Axx Series MCU SPI Demo(Each other as the master-slave) --*/
/* If you want to use the program or the program referenced in the  ----------*/
/* article, please specify in which data and procedures from STC    ----------*/
/*---------------------------------------------------------------------------------------*/

#include "reg51.h"

#define   FOSC              18432000L
#define   BAUD              (256 - FOSC / 32 / 115200)

typedef   unsigned char     BYTE;
typedef   unsigned int      WORD;
typedef   unsigned long     DWORD;

sfr       AUXR    =         0x8e;               //Auxiliary register

sfr       SPSTAT  =         0x84;                       //SPI status register
#define   SPIF              0x80                //SPSTAT.7
#define   WCOL              0x40                //SPSTAT.6
sfr       SPCTL   =         0x85;                       //SPI control register
#define   SSIG              0x80                //SPCTL.7
#define   SPEN              0x40                //SPCTL.6
#define   DORD              0x20                //SPCTL.5
#define   MSTR              0x10                //SPCTL.4
#define   CPOL              0x08                //SPCTL.3
#define   CPHA              0x04                //SPCTL.2
#define   SPDHH             0x00                //CPU_CLK/4
#define   SPDH              0x01                //CPU_CLK/16
#define   SPDL              0x02                //CPU_CLK/64
#define   SPDLL             0x03                //CPU_CLK/128
sfr       SPDAT   =         0x86;                       //SPI data register
sbit      SPISS   =         P1^3;               //SPI slave select, connect to slave' SS(P1.4) pin

void      InitUart();
void      InitSPI();
void      SendUart(BYTE dat);                   //send data to PC
BYTE      RecvUart();                           //receive data from PC
BYTE      SPISwap(BYTE dat);                    //swap SPI data between master
```

```
//////////////////////////////////////////////////////////

void main()
{
	InitUart();				//initial UART
	InitSPI();				//initial SPI

	while (1)
	{
		if (RI)
		{
			SPCTL = SPEN | MSTR;		//set as master
			SendUart(SPISwap(RecvUart()));
			SPCTL = SPEN;			//reset as slave
		}
		if (SPSTAT & SPIF)
		{
			SPSTAT = SPIF | WCOL; //clear SPI status
			SPDAT = SPDAT;		//mov data from receive buffer to send buffer
		}
	}
}

//////////////////////////////////////////////////////////

void InitUart()
{
	SCON = 0x5a;				//set UART mode as 8-bit variable baudrate
	TMOD = 0x20;				 //timer1 as 8-bit auto reload mode
	AUXR = 0x40;				//timer1 work at 1T mode
	TH1 = TL1 = BAUD;			//115200 bps
	TR1 = 1;
}

//////////////////////////////////////////////////////////

void InitSPI()
{
	SPDAT = 0;				//initial SPI data
	SPSTAT = SPIF | WCOL;			//clear SPI status
	SPCTL = SPEN;				//slave mode
}
```

```
//////////////////////////////////////////////////////////

void SendUart(BYTE dat)
{
        while (!TI);                    //wait pre-data sent
        TI = 0;                         //clear TI flag
        SBUF = dat;                     //send current data
}

//////////////////////////////////////////////////////////

BYTE RecvUart()
{
        while (!RI);                    //wait receive complete
        RI = 0;                         //clear RI flag
        return SBUF;                    //return receive data
}

//////////////////////////////////////////////////////////

BYTE SPISwap(BYTE dat)
{
        SPISS = 0;                      //pull low slave SS
        SPDAT = dat;                    //trigger SPI send
        while (!(SPSTAT & SPIF));       //wait send complete
        SPSTAT = SPIF | WCOL;           //clear SPI status
        SPISS = 1;                      //push high slave SS
        return SPDAT;                   //return received SPI data
}
```

2. Assemly code listing:

```
/*---------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------*/
/* --- STC12C5Axx Series MCU SPI Demo(Each other as the master-slave) --*/
/* If you want to use the program or the program referenced in the  ----------*/
/* article, please specify in which data and procedures from STC    ----------*/
/*---------------------------------------------------------------------------------------*/

AUXR    DATA    08EH                    ;Auxiliary register
SPSTAT  DATA    084H                    ;SPI status register
SPIF    EQU     080H                    ;SPSTAT.7
WCOL    EQU     040H                    ;SPSTAT.6
SPCTL   DATA    085H                    ;SPI control register
SSIG    EQU     080H                    ;SPCTL.7
SPEN    EQU     040H                    ;SPCTL.6
DORD    EQU     020H                    ;SPCTL.5
MSTR    EQU     010H                    ;SPCTL.4
CPOL    EQU     008H                    ;SPCTL.3
CPHA    EQU     004H                    ;SPCTL.2
SPDHH   EQU     000H                    ;CPU_CLK/4
SPDH    EQU     001H                    ;CPU_CLK/16
SPDL    EQU     002H                    ;CPU_CLK/64
SPDLL   EQU     003H                    ;CPU_CLK/128
SPDAT   DATA    086H                    ;SPI data register
SPISS   BIT     P1.3                    ;SPI slave select, connect to slave' SS(P1.4) pin

;///////////////////////////////////////////////////////////

        ORG     0000H
        LJMP    RESET
        ORG     0100H
RESET:
        LCALL   INIT_UART               ;initial UART
        LCALL   INIT_SPI                ;initial SPI
MAIN:
        JB      RI,     MASTER_MODE
```

```
SLAVE_MODE:
        MOV     A,          SPSTAT
        JNB     ACC.7,      MAIN
        MOV     SPSTAT,     #SPIF | WCOL                    ;clear SPI status
        MOV     SPDAT,      SPDAT                          ;return received SPI data
        SJMP    MAIN
MASTER_MODE:
        MOV     SPCTL,      #SPEN | MSTR                   ;set as master
        LCALL   RECV_UART                                  ;receive UART data from PC
        LCALL   SPI_SWAP            ;send it to slave, in the meantime, receive SPI data from slave
        LCALL   SEND_UART           ;send SPI data to PC
        MOV     SPCTL,      #SPEN   ;           ;reset as slave
        SJMP    MAIN

;///////////////////////////////////////////////////////////

INIT_UART:
        MOV     SCON,       #5AH    ;set UART mode as 8-bit variable baudrate
        MOV     TMOD,       #20H    ;timer1 as 8-bit auto reload mode
        MOV     AUXR,       #40H    ;timer1 work at 1T mode
        MOV     TL1,        #0FBH   ;115200 bps(256 - 18432000 / 32 / 115200)
        MOV     TH1,        #0FBH
        SETB    TR1
        RET

;///////////////////////////////////////////////////////////

INIT_SPI:
        MOV     SPDAT,      #0                             ;initial SPI data
        MOV     SPSTAT,     #SPIF | WCOL                   ;clear SPI status
        MOV     SPCTL,      #SPEN                          ;slave mode
        RET

;///////////////////////////////////////////////////////////

SEND_UART:
        JNB     TI,         $                              ;wait pre-data sent
        CLR     TI                                         ;clear TI flag
        MOV     SBUF,       A                              ;send current data
        RET
```

```
;/////////////////////////////////////////////////////////

RECV_UART:
        JNB     RI,     $                   ;wait receive complete
        CLR     RI                          ;clear RI flag
        MOV     A,      SBUF                ;return receive data
        RET
        RET

;/////////////////////////////////////////////////////////

SPI_SWAP:
        CLR     SPISS                       ;pull low slave SS
        MOV     SPDAT,  A                   ;trigger SPI send
WAIT:
        MOV     A,      SPSTAT
        JNB     ACC.7,  WAIT                ;wait send complete
        MOV     SPSTAT, #SPIF | WCOL        ;clear SPI status
        SETB    SPISS                       ;push high slave SS
        MOV     A,      SPDAT               ;return received SPI data
        RET

;/////////////////////////////////////////////////////////

        END
```

Chapter 12. IAP / EEPROM

The ISP in STC12C2052AD series makes it possible to update the user's application program and non-volatile application data (in IAP-memory) without removing the MCU chip from the actual end product. This useful capability makes a wide range of field-update applications possible. (Note ISP needs the loader program pre-programmed in the ISP-memory.) In general, the user needn't know how ISP operates because STC has provided the standard ISP tool and embedded ISP code in STC shipped samples.But, to develop a good program for ISP function, the user has to understand the architecture of the embedded flash.

The embedded flash consists of 20 pages. Each page contains 512 bytes. Dealing with flash, the user must erase it in page unit before writing (programming) data into it.Erasing flash means setting the content of that flash as FFh. Two erase modes are available in this chip. One is mass mode and the other is page mode. The mass mode gets more performance, but it erases the entire flash. The page mode is something performance less, but it is flexible since it erases flash in page unit. Unlike RAM's real-time operation, to erase flash or to write (program) flash often takes long time so to wait finish.

Furthermore, it is a quite complex timing procedure to erase/program flash. Fortunately, the STC12C2052AD series carried with convenient mechanism to help the user read/change the flash content. Just filling the target address and data into several SFR, and triggering the built-in ISP automation, the user can easily erase, read, and program the embedded flash.

The In-Application Program feature is designed for user to Read/Write nonvolatile data flash. It may bring great help to store parameters those should be independent of power-up and power-done action. In other words, the user can store data in data flash memory, and after he shutting down the MCU and rebooting the MCU, he can get the original value, which he had stored in.

The user can program the data flash according to the same way as ISP program, so he should get deeper understanding related to SFR ISP_DATA, ISP_ADDRL, ISP_ADDRH, ISP_CMD, ISP_TRIG, and ISP_CONTR.

12.1 IAP / EEPROM Special Function Registers

The following special function registers are related to the IAP/ISP/EEPROM operation. All these registers can be accessed by software in the user's application program.

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
ISP_DATA	ISP/IAP Flash Data Register	E2H									1111 1111B
ISP_ADDRH	ISP/IAP Flash Address High	E3H									0000 0000B
ISP_ADDRL	ISP/IAP Flash Address Low	E4H									0000 0000B
ISP_CMD	ISP/IAP Flash Command Register	E5H	-	-	-	-	-	-	MS1	MS0	xxxx x000B
ISP_TRIG	ISP/IAP Flash Command Trigger	E6H									xxxx xxxxB
ISP_CONTR	ISP/IAP Control Register	E7H	ISPEN	SWBS	SWRST	CMD_FAIL	-	WT2	WT1	WT0	0000 1000B

1. ISP/IAP Flash Data Register : ISP_DATA (Address: E2H, Non bit-addressable)

ISP_DATA is the data port register for ISP/IAP operation. The data in ISP_DATA will be written into the desired address in operating ISP/IAP write and it is the data window of readout in operating ISP/IAP read.

2. ISP/IAP Flash Address Registers : ISP_ADDRH and ISP_ADDRL

ISP_ADDRH (address:E3H) is the high-byte address port for all ISP/IAP modes.

ISP_ADDRH[7:5] must be cleared to 000, if one bit of ISP_ADDRH[7:5] is set, the IAP/ISP write function must fail.

ISP_ADDRL (address:E4H) is the low port for all ISP/IAP modes. In page erase operation, it is ignored.

3. ISP/IAP Flash Command Register : ISP_CMD (Non bit -addressable)

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
ISP_CMD	E5H	name	-	-	-	-	-	-	MS1	MS0

B7~B2: Reserved.

MS1, MS0 : ISP/IAP operating mode selection. IAP_CMD is used to select the flash mode for performing numerous ISP/IAP function or used to access protected SFRs.

0, 0 : Standby

0, 1 : Data Flash/EEPROM read.

1, 0 : Data Flash/EEPROM program.

1, 1 : Data Flash/EEPROM page erase.

4. ISP/IAP Flash Command Trigger Register : ISP_TRIG (Address: E6H, Non bit -addressable)

ISP_TRIG is the command port for triggering ISP/IAP activity and protected SFRs access. If ISP_TRIG is filled with sequential 0x46h, 0xB9h and if ISPEN(ISP_CONTR.7) = 1, ISP/IAP activity or protected SFRs access will triggered.

5. ISP/IAP Control Register : ISP_CONTR (Non bit-addressable)

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

ISPEN : ISP/IAP operation enable.

0 : Global disable all ISP/IAP program/erase/read function.

1 : Enable ISP/IAP program/erase/read function.

SWBS: software boot selection control.

0 : Boot from main-memory after reset.

1 : Boot from ISP memory after reset.

SWRST: software reset trigger control.

0 : No operation

1 : Generate software system reset. It will be cleared by hardware automatically.

CMD_FAIL: Command Fail indication for ISP/IAP operation.

0 : The last ISP/IAP command has finished successfully.

1 : The last ISP/IAP command fails. It could be caused since the access of flash memory was inhibited.

B3: Reserved. Software must write “0” on this bit when ISP_CONTR is written.

WT2~WT0 : Waiting time selection while flash is busy.

Setting wait times			CPU wait times			
WT2	WT1	WT0	Read (2 System clocks)	Program (=55uS)	Sector Erase (=21mS)	Recommended System Clock Frequency (MHz)
1	1	1	2 SYSclks	55 SYSclks	21012 SYSclks	≤1MHz
1	1	0	2 SYSclks	110 SYSclks	42024 SYSclks	≤ 2MHz
1	0	1	2 SYSclks	165 SYSclks	63036 SYSclks	≤ 3MHz
1	0	0	2 SYSclks	330 SYSclks	126072 SYSclks	≤ 6MHz
0	1	1	2 SYSclks	660 SYSclks	252144 SYSclks	≤ 12MHz
0	1	0	2 SYSclks	1100 SYSclks	420240 SYSclks	≤ 20MHz
0	0	1	2 SYSclks	1320 SYSclks	504288 SYSclks	≤ 24MHz
0	0	0	2 SYSclks	1760 SYSclks	672384 SYSclks	≤ 30MHz

Note: Software reset actions could reset other SFR,but it never influences bits ISPEN and SWBS.The ISPEN and SWBS. The ISPEN and SWBS only will be reset by power-up action, while not software reset.

6. When the operation voltage is too low, EEPROM / IAP function should be disabled

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

LVDF : Pin Low-Voltage Flag. Once low voltage condition is detected (VCC power is lower than LVD voltage), it is set by hardware (and should be cleared by software).

12.2 STC12C2052AD series Internal EEPROM Allocation Table

STC12C2052AD series microcontroller's Data Flash (internal available EEPROM) address (and program space is separate) : if the application area of IAP write Data/erase sector of the action, the statements will be ignore and continue to the next one. Program in user application area (AP area), only operate IAP/ISP on Data Flash (EEPROM)

STC12C5052AD and STC12LE5052AD are excepted, this several types in the application area can modify the application

STC12C2052AD series MCU internal EEPROM Selection Table STC12LE2052AD series MCU internal EEPROM Selection Table				
Type	**EEPROM (Byte)**	**Sector Numbers**	**Begin_Sector Begin_Address**	**End_Sector End_Address**
STC12C1052AD/ STC12LE1052AD	10K	20	0000h	27FFh
STC12C2052AD/ STC12LE2052AD	10K	20	0000h	27FFh
STC12C3052AD/ STC12LE3052AD	10K	20	0000h	27FFh
STC12C4052AD/ STC12LE4052AD	10K	20	0000h	27FFh
The following series are special. User can modify the application the application area, all flash area can be modified as EEPROM				
STC12C5052AD/ STC12LE5052AD	-	10	0000h	27FFh

STC12C2052AD series MCU address reference table in detail (512 bytes per sector) STC12LE2052AD series MCU address reference table in detail (512 bytes per sector)								
Sector 1		Sector 2		Sector 3		Sector 4		Each sector 512 byte
Start	End	Start	End	Start	End	Start	End	
0000H	01FFH	0200H	03FFH	0400H	05FFH	0600H	07FFH	
Sector 5		Sector 6		Sector7		Sector 8		
Start	End	Start	End	Start	End	Start	End	
0800H	09FFH	0A00H	0BFFH	0C00H	0DFFH	0E00H	0FFFH	Suggest the same times modified data in the same sector, each times modified data in different sectors, don't have to use full, of course, it was all to use
Sector 9		Sector 10		Sector 11		Sector 12		
Start	End	Start	End	Start	End	Start	End	
1000H	11FFH	1200H	13FFH	1400H	15FFH	1600H	17FFH	
Sector 13		Sector 14		Sector 15		Sector 16		
Start	End	Start	End	Start	End	Start	End	
1800H	19FFH	1A00H	1BFFH	1C000H	1DFFH	1E00H	1FFFH	
Sector 17		Sector 18		Sector 19		Sector 20		
Start	End	Start	End	Start	End	Start	End	
2000H	21FFH	2200H	23FFH	2400H	25FFH	2600H	27FFH	

12.3 IAP/EEPROM Assembly Language Program Introduction

```
; /*It is decided by the assembler/compiler used by users that whether the SFRs addresses are declared by the
DATA or the EQU directive*/
        ISP_DATA        DATA    0E2H    or      ISP_DATA        EQU     0E2H
        ISP_ADDRH       DATA    0E3H    or      ISP_ADDRH       EQU     0E3H
        ISP_ADDRL       DATA    0E4H    or      ISP_ADDRL       EQU     0E4H
        ISP_CMD         DATA    0E5H    or      ISP_CMD         EQU     0E5H
        ISP_TRIG        DATA    0E6H    or      ISP_TRIG        EQU     0E6H
        ISP_CONTR       DATA    0E7H    or      ISP_CONTR       EQU     0E7H

;/*Define ISP/IAP/EEPROM command and wait time*/
        ISP_IAP_BYTE_READ               EQU     1       ;Byte-Read
        ISP_IAP_BYTE_PROGRAM            EQU     2       ;Byte-Program
        ISP_IAP_SECTOR_ERASE            EQU     3       ;Sector-Erase
        WAIT_TIME                       EQU     0       ;Set wait time

;/*Byte-Read*/
        MOV     ISP_ADDRH,      #BYTE_ADDR_HIGH         ;Set ISP/IAP/EEPROM address high
        MOV     ISP_ADDRL,      #BYTE_ADDR_LOW          ;Set ISP/IAP/EEPROM address low
        MOV     ISP_CONTR,      #WAIT_TIME              ;Set wait time
        ORL     ISP_CONTR,      #10000000B              ;Open ISP/IAP function
        MOV     ISP_CMD,        #ISP_IAP_BYTE_READ      ;Set ISP/IAP Byte-Read command
        MOV     ISP_TRIG,       #46H                    ;Send trigger command1 (0x46)
        MOV     ISP_TRIG,       #0B9H                   ;Send trigger command2 (0xB9)
        NOP                     ;CPU will hold here until ISP/IAP/EEPROM operation complete
        MOV     A,      ISP_DATA        ;Read ISP/IAP/EEPROM data

;/*Disable ISP/IAP/EEPROM function, make MCU in a safe state*/
        MOV     ISP_CONTR,      #00000000B              ;Close ISP/IAP/EEPROM function
        MOV     ISP_CMD,        #00000000B              ;Clear ISP/IAP/EEPROM command
        ;MOV    ISP_TRIG,       #00000000B              ;Clear trigger register to prevent mistrigger
        ;MOV    ISP_ADDRH,      #0FFH                   ;Move 00 into address high-byte unit,
                                                        ;Data ptr point to non-EEPROM area
        ;MOV    ISP_ADDRL,      #0FFH                   ;Move 00 into address low-byte unit,
                                                        ;prevent misuse

;/*Byte-Program, if the byte is null(0FFH), it can be programmed; else, MCU must operate Sector-Erase firstly,
and then can operate Byte-Program.*/
        MOV     ISP_DATA,       #ONE_DATA               ;Write ISP/IAP/EEPROM data
        MOV     ISP_ADDRH,      #BYTE_ADDR_HIGH         ;Set ISP/IAP/EEPROM address high
        MOV     ISP_ADDRL,      #BYTE_ADDR_LOW          ;Set ISP/IAP/EEPROM address low
        MOV     ISP_CONTR,      #WAIT_TIME              ;Set wait time
        ORL     ISP_CONTR,      #10000000B              ;Open ISP/IAP function
        MOV     ISP_CMD,        #ISP_IAP_BYTE_READ      ;Set ISP/IAP Byte-Read command
        MOV     ISP_TRIG,       #46H                    ;Send trigger command1 (0x46)
        MOV     ISP_TRIG,       #0B9H                   ;Send trigger command2 (0xb9)
        NOP                     ;CPU will hold here until ISP/IAP/EEPROM operation complete
```

```
;/*Disable ISP/IAP/EEPROM function, make MCU in a safe state*/
        MOV     ISP_CONTR,      #00000000B      ;Close ISP/IAP/EEPROM function
        MOV     ISP_CMD,        #00000000B      ;Clear ISP/IAP/EEPROM command
        ;MOV    ISP_TRIG,       #00000000B      ;Clear trigger register to prevent mistrigger
        ;MOV    ISP_ADDRH,      #FFH            ;Move 00H into address high-byte unit,
                                                ;Data ptr point to non-EEPROM area
        ;MOV    ISP_ADDRL,      #0FFH           ;Move 00H into address low-byte unit,
                                                ;prevent misuse

;/*Erase one sector area, there is only Sector-Erase instead of Byte-Erase, every sector area account for 512 bytes*/
        MOV     ISP_ADDRH,      #SECTOT_FIRST_BYTE_ADDR_HIGH
                                                ;Set the sector area starting address high
        MOV     ISP_ADDRL,      #SECTOT_FIRST_BYTE_ADDR_LOW
                                                ;Set the sector area starting address low
        MOV     ISP_CONTR,      #WAIT_TIME      ;Set wait time
        ORL     ISP_CONTR,      #10000000B      ;Open ISP/IAP function
        MOV     ISP_CMD,        #ISP_IAP_SECTOR_ERASE           ;Set Sectot-Erase command
        MOV     ISP_TRIG,       #46H            ;Send trigger command1 (0x46)
        MOV     ISP_TRIG,       #0B9H           ;Send trigger command2 (0xb9)
        NOP                     ;CPU will hold here until ISP/IAP/EEPROM operation complete

;/*Disable ISP/IAP/EEPROM function, make MCU in a safe state*/
        MOV     ISP_CONTR,      #00000000B      ;Close ISP/IAP/EEPROM function
        MOV     ISP_CMD,        #00000000B      ;Clear ISP/IAP/EEPROM command
        ;MOV    ISP_TRIG,       #00000000B      ;Clear trigger register to prevent mistrigger
        ;MOV    ISP_ADDRH,      #0FFH           ;Move 00H into address high-byte unit,
                                                ; Data ptr point to non-EEPROM area
        ;MOV    ISP_ADDRL,      #0FFH           ;Move 00H into address low-byte unit,
                                                ;prevent misuse
```

Little common sense: (STC MCU Data Flash use as EEPROM function)

Three basic commands -- bytes read, byte programming, the sector erased

Byte programming: "1" write "1" or "0", will "0" write "0".Just FFH can byte programming. If the byte not FFH, you must erase the sector , because only the "sectors erased" to put "0" into "1".

Sector erased: only "sector erased" will also be a "0" erased for "1".

Big proposal:

1. The same times modified data in the same sector, not the same times modified data in other sectors, won't have to read protection.

2. If a sector with only one byte, that's real EEPROM, STC MCU Data Flash faster than external EEPROM, read a byte/many one byte programming is about 2 clock / 55uS.

3. If in a sector of storing a large amounts of data, a only need to modify one part of a byte, or when the other byte don't need to modify data must first read on STC MCU, then erased RAM the whole sector, again will need to keep data and need to amend data in bytes written back to this sector section literally only bytes written orders (without continuous bytes, write command). Then each sector use bytes are using the less the convenient (not need read a lot of maintained data).

Frequently asked questions:

1. IAP instructions after finishing, address is automatically "add 1" or "minus 1"?
   Answer: not

2. Send 46 and B9 after IAP ordered the trigger whether to have sent 46 and B9 trigger?
   Answer: yes

12.4 EEPROM Demo Program (C and ASM)

1. C Code Listing

```
/*---------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------*/
/* --- STC 1T Series MCU ISP/IAP/EEPROM Demo -------------------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*---------------------------------------------------------------------------*/

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

typedef unsigned char BYTE;
typedef unsigned int WORD;

/*Declare SFR associated with the IAP */
sfr         IAP_DATA        =   0xE2;           //Flash data register
sfr         IAP_ADDRH       =   0xE3;           //Flash address HIGH
sfr         IAP_ADDRL       =   0xE4;           //Flash address LOW
sfr         IAP_CMD         =   0xE5;           //Flash command register
sfr         IAP_TRIG        =   0xE6;           //Flash command trigger
sfr         IAP_CONTR       =   0xE7;           //Flash control register

/*Define ISP/IAP/EEPROM command*/
#define    CMD_IDLE         0                   //Stand-By
#define    CMD_READ         1                   //Byte-Read
#define    CMD_PROGRAM      2                   //Byte-Program
#define    CMD_ERASE        3                   //Sector-Erase

/*Define ISP/IAP/EEPROM operation const for IAP_CONTR*/
//#define  ENABLE_IAP       0x80                //if SYSCLK<30MHz
//#define  ENABLE_IAP       0x81                //if SYSCLK<24MHz
#define    ENABLE_IAP       0x82                //if SYSCLK<20MHz
//#define  ENABLE_IAP       0x83                //if SYSCLK<12MHz
//#define  ENABLE_IAP       0x84                //if SYSCLK<6MHz
//#define  ENABLE_IAP       0x85                //if SYSCLK<3MHz
//#define  ENABLE_IAP       0x86                //if SYSCLK<2MHz
//#define  ENABLE_IAP       0x87                //if SYSCLK<1MHz

//Start address for STC12C2052AD series EEPROM
#define    IAP_ADDRESS      0x0400

void Delay(BYTE n);
void IapIdle();
BYTE IapReadByte(WORD addr);
```

```
void IapProgramByte(WORD addr, BYTE dat);
void IapEraseSector(WORD addr);

void main()
{
	WORD i;

	P1 = 0xfe;				//1111,1110 System Reset OK
	Delay(10);				//Delay
	IapEraseSector(IAP_ADDRESS);		//Erase current sector
	for (i=0; i<512; i++)			//Check whether all sector data is FF
	{
		if (IapReadByte(IAP_ADDRESS+i) != 0xff)
		goto Error;			//If error, break
	}
	P1 = 0xfc;				//1111,1100 Erase successful
	Delay(10);				//Delay
	for (i=0; i<512; i++)			//Program 512 bytes data into data flash
	{
		IapProgramByte(IAP_ADDRESS+i, (BYTE)i);
	}
	P1 = 0xf8;				//1111,1000 Program successful
	Delay(10);				//Delay
	for (i=0; i<512; i++)			//Verify 512 bytes data
	{
		if (IapReadByte(IAP_ADDRESS+i) != (BYTE)i)
		goto Error;			//If error, break
	}
	P1 = 0xf0;				//1111,0000 Verify successful
	while (1);
Error:
	P1 &= 0x7f;				//0xxx,xxxx IAP operation fail
	while (1);
}

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

	while (n--)
	{
		x = 0;
		while (++x);
	}
}
```

```
/*----------------------------
Disable ISP/IAP/EEPROM function
Make MCU in a safe state
----------------------------*/
void IapIdle()
{
	IAP_CONTR = 0;			//Close IAP function
	IAP_CMD = 0;			//Clear command to standby
	IAP_TRIG = 0;			//Clear trigger register
	IAP_ADDRH = 0x80;		//Data ptr point to non-EEPROM area
	IAP_ADDRL = 0;			//Clear IAP address to prevent misuse
}

/*----------------------------
Read one byte from ISP/IAP/EEPROM area
Input: addr (ISP/IAP/EEPROM address)
Output:Flash data
----------------------------*/
BYTE IapReadByte(WORD addr)
{
	BYTE dat;			//Data buffer

	IAP_CONTR = ENABLE_IAP;		//Open IAP function, and set wait time
	IAP_CMD = CMD_READ;		//Set ISP/IAP/EEPROM READ command
	IAP_ADDRL = addr;		//Set ISP/IAP/EEPROM address low
	IAP_ADDRH = addr >> 8;		//Set ISP/IAP/EEPROM address high
	IAP_TRIG = 0x46;		//Send trigger command1 (0x46)
	IAP_TRIG = 0xb9;		//Send trigger command2 (0xb9)
	_nop_();			//MCU will hold here until ISP/IAP/EEPROM
					//operation complete
	dat = IAP_DATA;			//Read ISP/IAP/EEPROM data
	IapIdle();			//Close ISP/IAP/EEPROM function

	return dat;			//Return Flash data
}

/*----------------------------
Program one byte to ISP/IAP/EEPROM area
Input: addr (ISP/IAP/EEPROM address)
     dat (ISP/IAP/EEPROM data)
Output:-
----------------------------*/
```

```
void IapProgramByte(WORD addr, BYTE dat)
{
	IAP_CONTR = ENABLE_IAP;		//Open IAP function, and set wait time
	IAP_CMD = CMD_PROGRAM;		//Set ISP/IAP/EEPROM PROGRAM command
	IAP_ADDRL = addr;		//Set ISP/IAP/EEPROM address low
	IAP_ADDRH = addr >> 8;		//Set ISP/IAP/EEPROM address high
	IAP_DATA = dat;			//Write ISP/IAP/EEPROM data
	IAP_TRIG = 0x46;		//Send trigger command1 (0x46)
	IAP_TRIG = 0xb9;		//Send trigger command2 (0xb9)
	_nop_();			//MCU will hold here until ISP/IAP/EEPROM
					//operation complete
	IapIdle();
}

/*------------------------------
Erase one sector area
Input: addr (ISP/IAP/EEPROM address)
Output:-
------------------------------*/
void IapEraseSector(WORD addr)
{
	IAP_CONTR = ENABLE_IAP;		//Open IAP function, and set wait time
	IAP_CMD = CMD_ERASE;		//Set ISP/IAP/EEPROM ERASE command
	IAP_ADDRL = addr;		//Set ISP/IAP/EEPROM address low
	IAP_ADDRH = addr >> 8;		//Set ISP/IAP/EEPROM address high
	IAP_TRIG = 0x46;		//Send trigger command1 (0x46)
	IAP_TRIG = 0xb9;		//Send trigger command2 (0xb9)
	_nop_();			//MCU will hold here until ISP/IAP/EEPROM
					//operation complete
	IapIdle();
}
```

2. Assembly Code Listing

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------*/
;/* --- STC 1T Series MCU ISP/IAP/EEPROM Demo -------------------------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*---------------------------------------------------------------------------*/

;/*Declare SFRs associated with the IAP */
IAP_DATA        EQU     0E2H            ;Flash data register
IAP_ADDRH       EQU     0E3H            ;Flash address HIGH
IAP_ADDRL       EQU     0E4H            ;Flash address LOW
IAP_CMD         EQU     0E5H            ;Flash command register
IAP_TRIG        EQU     0E6H            ;Flash command trigger
IAP_CONTR       EQU     0E7H            ;Flash control register

;/*Define ISP/IAP/EEPROM command*/
CMD_IDLE        EQU     0               ;Stand-By
CMD_READ        EQU     1               ;Byte-Read
CMD_PROGRAM EQU     2               ;Byte-Program
CMD_ERASE       EQU     3               ;Sector-Erase

;/*Define ISP/IAP/EEPROM operation const for IAP_CONTR*/
;ENABLE_IAP     EQU     80H             ;if SYSCLK<30MHz
;ENABLE_IAP     EQU     81H             ;if SYSCLK<24MHz
ENABLE_IAP      EQU     82H             ;if SYSCLK<20MHz
;ENABLE_IAP     EQU     83H             ;if SYSCLK<12MHz
;ENABLE_IAP     EQU     84H             ;if SYSCLK<6MHz
;ENABLE_IAP     EQU     85H             ;if SYSCLK<3MHz
;ENABLE_IAP     EQU     86H             ;if SYSCLK<2MHz
;ENABLE_IAP     EQU     87H             ;if SYSCLK<1MHz

;//Start address for STC12C2052AD series EEPROM
IAP_ADDRESS EQU 0000H
;-----------------------------------------
        ORG     0000H
        LJMP    MAIN
;-----------------------------------------
        ORG     0100H
MAIN:
        MOV     P1,     #0FEH           ;1111,1110 System Reset OK
        LCALL   DELAY                   ;Delay
```

```
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS        ;Set ISP/IAP/EEPROM address
        LCALL   IAP_ERASE                   ;Erase current sector
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS        ;Set ISP/IAP/EEPROM address
        MOV     R0,     #0                  ;Set counter (512)
        MOV     R1,     #2
CHECK1:                                     ;Check whether all sector data is FF
        LCALL   IAP_READ                    ;Read Flash
        CJNE    A,      #0FFH,  ERROR       ;If error, break
        INC     DPTR                        ;Inc Flash address
        DJNZ    R0,     CHECK1              ;Check next
        DJNZ    R1,     CHECK1              ;Check next
;-------------------------------
        MOV     P1,     #0FCH               ;1111,1100 Erase successful
        LCALL   DELAY                       ;Delay
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS        ;Set ISP/IAP/EEPROM address
        MOV     R0,     #0                  ;Set counter (512)
        MOV     R1,     #2
        MOV     R2,     #0                  ;Initial test data
NEXT:                                       ;Program 512 bytes data into data flash
        MOV     A,      R2                  ;Ready IAP data
        LCALL   IAP_PROGRAM                 ;Program flash
        INC     DPTR                        ;Inc Flash address
        INC     R2                          ;Modify test data
        DJNZ    R0,     NEXT                ;Program next
        DJNZ    R1,     NEXT                ;Program next
;-------------------------------
        MOV     P1,     #0F8H               ;1111,1000 Program successful
        LCALL   DELAY                       ;Delay
;-------------------------------
        MOV     DPTR,   #IAP_ADDRESS        ;Set ISP/IAP/EEPROM address
        MOV     R0,     #0                  ;Set counter (512)
        MOV     R1,     #2
        MOV     R2,     #0
CHECK2:                                     ;Verify 512 bytes data
        LCALL   IAP_READ                    ;Read Flash
        CJNE    A,  2,  ERROR               ;If error, break
        INC     DPTR                        ;Inc Flash address
        INC     R2                          ;Modify verify data
        DJNZ    R0,     CHECK2              ;Check next
        DJNZ    R1,     CHECK2              ;Check next
;-------------------------------
        MOV     P1,     #0F0H               ;1111,0000 Verify successful
        SJMP    $
;-------------------------------
```

```
ERROR:
        MOV     P0,     R0
        MOV     P2,     R1
        MOV     P3,     R2
        CLR     P1.7                    ;0xxx,xxxx IAP operation fail
        SJMP    $

;/*----------------------------
;Software delay function
;----------------------------*/
DELAY:
        CLR     A
        MOV     R0,     A
        MOV     R1,     A
        MOV     R2,     #20H
DELAY1:
        DJNZ    R0,     DELAY1
        DJNZ    R1,     DELAY1
        DJNZ    R2,     DELAY1
        RET

;/*----------------------------
;Disable ISP/IAP/EEPROM function
;Make MCU in a safe state
;----------------------------*/
IAP_IDLE:
        MOV     IAP_CONTR,      #0              ;Close IAP function
        MOV     IAP_CMD,        #0              ;Clear command to standby
        MOV     IAP_TRIG,       #0              ;Clear trigger register
        MOV     IAP_ADDRH,      #80H            ;Data ptr point to non-EEPROM area
        MOV     IAP_ADDRL,      #0              ;Clear IAP address to prevent misuse
        RET

;/*----------------------------
;Read one byte from ISP/IAP/EEPROM area
;Input: DPTR(ISP/IAP/EEPROM address)
;Output:ACC (Flash data)
;----------------------------*/
IAP_READ:
        MOV     IAP_CONTR,      #ENABLE_IAP     ;Open IAP function, and set wait time
        MOV     IAP_CMD,        #CMD_READ       ;Set ISP/IAP/EEPROM READ command
        MOV     IAP_ADDRL,      DPL             ;Set ISP/IAP/EEPROM address low
        MOV     IAP_ADDRH,      DPH             ;Set ISP/IAP/EEPROM address high
        MOV     IAP_TRIG,       #46H            ;Send trigger command1 (0x46)
        MOV     IAP_TRIG,       #0B9H           ;Send trigger command2 (0xb9)
        NOP                     ;MCU will hold here until ISP/IAP/EEPROM operation complete
        MOV     A,      IAP_DATA                ;Read ISP/IAP/EEPROM data
        LCALL   IAP_IDLE                        ;Close ISP/IAP/EEPROM function
        RET
```

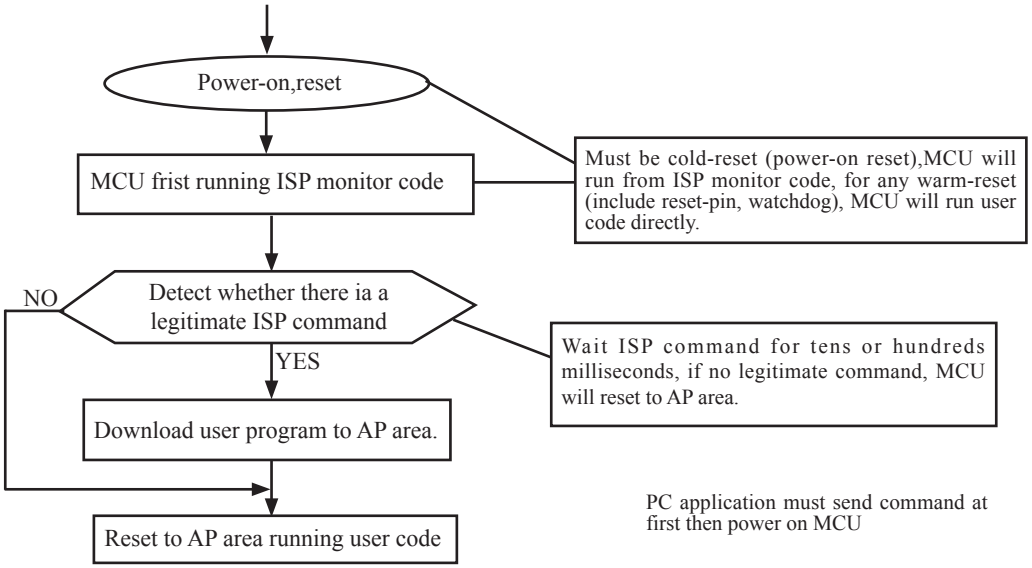
```
;/*------------------------------
;Program one byte to ISP/IAP/EEPROM area
;Input: DPAT(ISP/IAP/EEPROM address)
;ACC (ISP/IAP/EEPROM data)
;Output:-
;------------------------------*/
IAP_PROGRAM:
        MOV     IAP_CONTR,      #ENABLE_IAP     ;Open IAP function, and set wait time
        MOV     IAP_CMD,        #CMD_PROGRAM    ;Set ISP/IAP/EEPROM PROGRAM command
        MOV     IAP_ADDRL,      DPL             ;Set ISP/IAP/EEPROM address low
        MOV     IAP_ADDRH,      DPH             ;Set ISP/IAP/EEPROM address high
        MOV     IAP_DATA,       A               ;Write ISP/IAP/EEPROM data
        MOV     IAP_TRIG,       #46H            ;Send trigger command1 (0x46)
        MOV     IAP_TRIG,       #0B9H           ;Send trigger command2 (0xb9)
        NOP                     ;MCU will hold here until ISP/IAP/EEPROM operation complete
        LCALL   IAP_IDLE                        ;Close ISP/IAP/EEPROM function
        RET

;/*------------------------------
;Erase one sector area
;Input: DPTR(ISP/IAP/EEPROM address)
;Output:-
;------------------------------*/
IAP_ERASE:
        MOV     IAP_CONTR,      #ENABLE_IAP     ;Open IAP function, and set wait time
        MOV     IAP_CMD,        #CMD_ERASE      ;Set ISP/IAP/EEPROM ERASE command
        MOV     IAP_ADDRL,      DPL             ;Set ISP/IAP/EEPROM address low
        MOV     IAP_ADDRH,      DPH             ;Set ISP/IAP/EEPROM address high
        MOV   IAP_TRIG,#46H                     ;Send trigger command1 (0x46)
        MOV   IAP_TRIG,#0B9H                    ;Send trigger command2 (0xb9)
        NOP                     ;MCU will hold here until ISP/IAP/EEPROM operation complete
        LCALL   IAP_IDLE                        ;Close ISP/IAP/EEPROM function
        RET

        END
```

Chapter 13. STC12 series Development/Programming Tool

13.1 In-System-Programming (ISP) principle

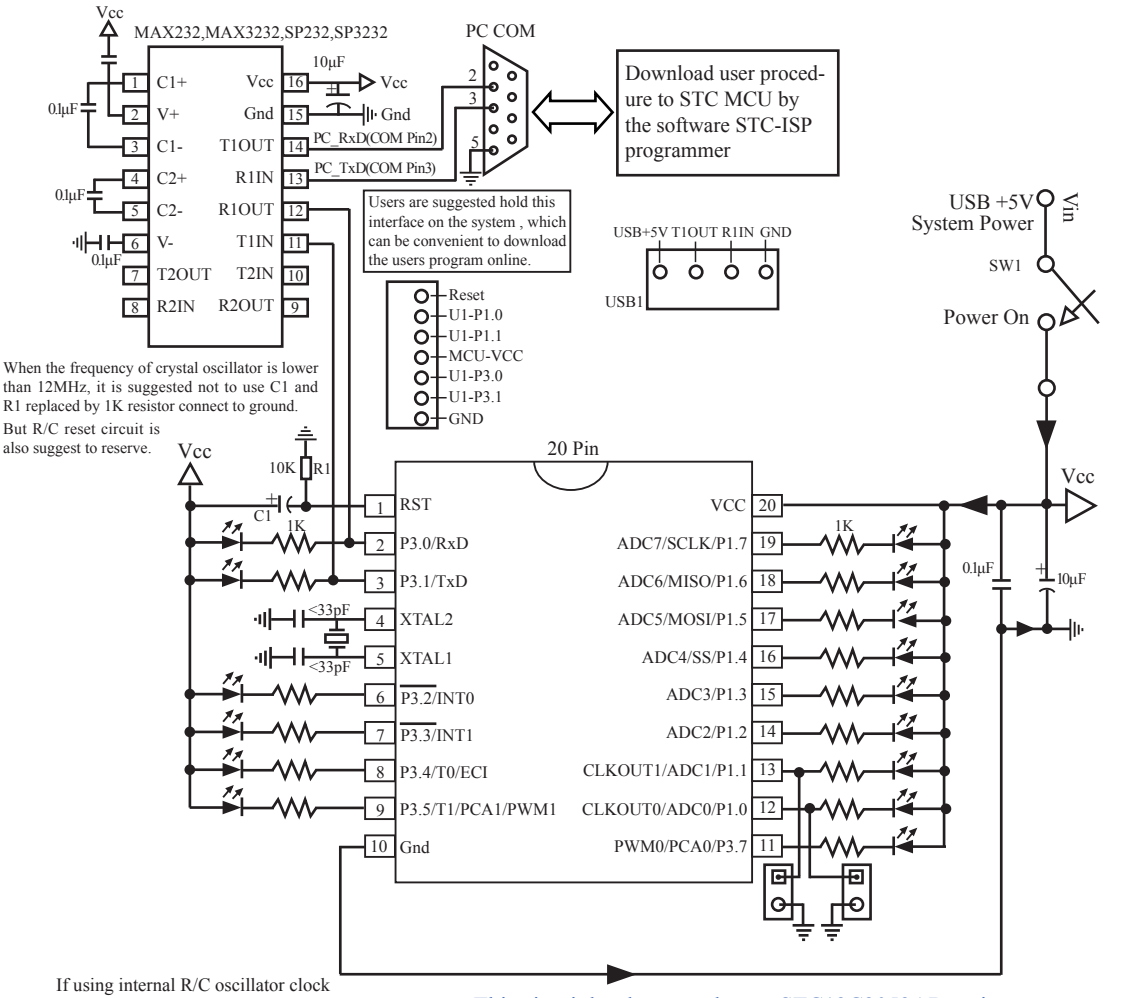
If need download code into STC12C2052AD series, P1.0 and P1.1 pin must be connected to GND

If you chose the "Next program code, P1.0/1.1 need=0/0" option, then the next time you need to re-download the program, first of all must be connected P1.0 and P1.1 to GND



PC application must send command at first then power on MCU

13.2 STC12C2052AD series Typical Application Circuit for ISP



If using internal R/C oscillator clock (4MHz ~ 8MHz, manufacturing error), XTAL1 and XTAL2 pin should be floated.

If External clock frequency is higher than 33MHz, it is recommended to directly use external active crystals which clock are input from XTAL1 pin and XTAL2 pin must be floated.

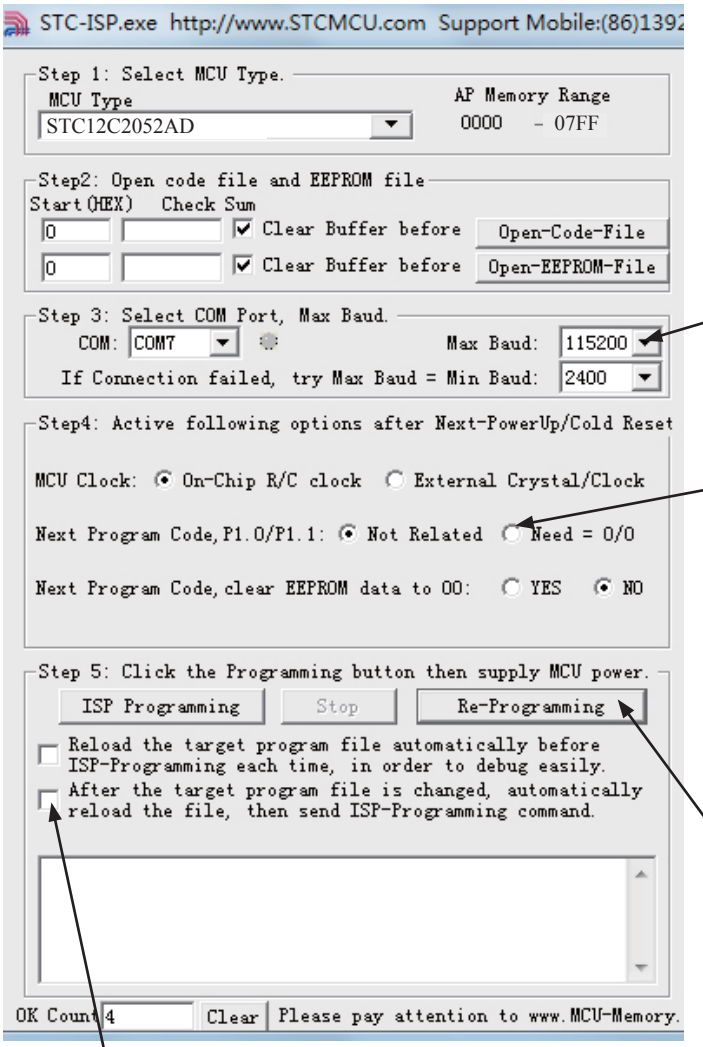
This circuit has been made as a STC12C2052AD series microcontroller ISP download programming tool

Users in their target system, such as the P3.0/P3.1 through the RS-232 level shifter connected to the computer after the conversion of ordinary RS-232 serial port to connect the system programming / upgrading client software. If the user panel recommended no RS-232 level converter, should lead to a socket, with Gnd/P3.1/P3.0/Vcc four signal lines, so that the user system can be programmed directly. Of course, if the six signal lines can lead to Gnd/P3.1/P3.0/Vcc/P1.1/P1.0 as well, because you can download the program by P1.0/P1.1 ISP ban. If you can Gnd/P3.1/P3.0/Vcc/P1.1/P1.0/Reset seven signal lines leads to better, so you can easily use "offline download board (no computer)" .

ISP programming on the Theory and Application Guide to see "STC12C2052AD Series MCU Development / Programming Tools Help"section. In addition, we have standardized programming download tool, the user can then program into the goal in the above systems, you can borrow on top of it RS-232 level shifter connected to the computer to download the program used to do. Programming a chip roughly be a few seconds, faster than the ordinary universal programmer much faster, there is no need to buy expensive third-party programmer?.

PC STC-ISP software downloaded from the website

13.3 PC side application usage



According to actual situation, the user selects the appropriate maximum baud rate

In practice, if P3.0/ P3.1 already connected to a RS232/RS485 or other equipment, it is recommended that selection P1.0 / P1.1 = 0/0 can download options

Press this button when mass production

All new settings are valid in the next power-on.

Enable the option in debugging stage

Step1 : Select MCU type (E.g. STC12C2052AD)
Step2 : Load user program code (*.bin or *.hex)
Setp3 : Select the serial port you are using
Setp4 : Config the hardware option
Step5 : Press “ISP programming” or “Re-Programming” button to download user program
NOTE : Must press “ISP programming” or “Re-Programming” button first, then power on MCU, otherwise will cannot download.

About hardware connection

1. MCU RXD (P3.0) ---- RS232 ---- PC COM port TXD (Pin3)
2. MCU TXD (P3.1) ---- RS232 ---- PC COM port RXD (Pin2)
3. MCU GNG-------PC COM port GND (Pin5)
4. RS232 : You can select STC232 / STC3232 / MAX232 / MAX3232 / …

Using a demo board as a programmer

STC-ISP ver3.0A PCB can be welded into three kinds of circuits, respectively, support the STC's 16/20/28/32 pins MCU, the back plate of the download boards are affixed with labels,users need to pay special attention to. All the download board is welded 40-pin socket, the socket’s 20-pin is ground line, all types of MCU should be put on the socket according to the way of alignment with the ground. The method of programming user code using download board as follow:

1. According to the type of MCU choose supply voltage,
   A. For 5V MCU, using jumper JP1 to connect MCU-VCC to +5V pin
   B. For 3V MCU, using jumper JP1 to connect MCU-VCC to +3.3V pin
2. Download cable (Provide by STC)
   A. Connect DB9 serial connector to the computer's RS-232 serial interface
   B. Plug the USB interface at the same side into your computer's USB port for power supply
   C. Connect the USB interface at the other side into STC download board
3. Other interfaces do not need to connect.
4. In a non-pressed state to SW1, and MCU-VCC power LED off.
5. For SW3
   P1.0/P1.1 = 1/1 when SW3 is non-pressed
   P1.0/P1.1 = 0/0 when SW3 is pressed
   If you have select the “Next program code, P1.0/P1.1 Need = 0/0” option, then SW3 must be in a pressed state
6. Put target MCU into the U1 socket, and locking socket
7. Press the “Download” button in the PC side application
8. Press SW1 switch in the download board
9. Close the demo board power supply and remove the MCU after download successfully.

13.4 Compiler / Assembler Programmer and Emulator

About Compiler/Assembler

Any traditional compiler / assembler and the popular Keil are suitable for STC MCU. For selection MCU body, the traditional compiler / assembler, you can choose Intel's 8052 / 87C52 / 87C52 / 87C58 or Philips's P87C52 / P87C54/P87C58 in the traditional environment, in Keil environment, you can choose the types in front of the proposed or download the STC chips database file (STC.CDB) from the STC official website.

About Programmer

You can use the STC specific ISP programmer. (Can be purchased from the STC or apply for free sample). Programmer can be used as demo board

About Emulator

We do not provite specific emulator now. If you have a traditional 8051 emulator, you can use it to simulate STC MCU's some 8052 basic functions.

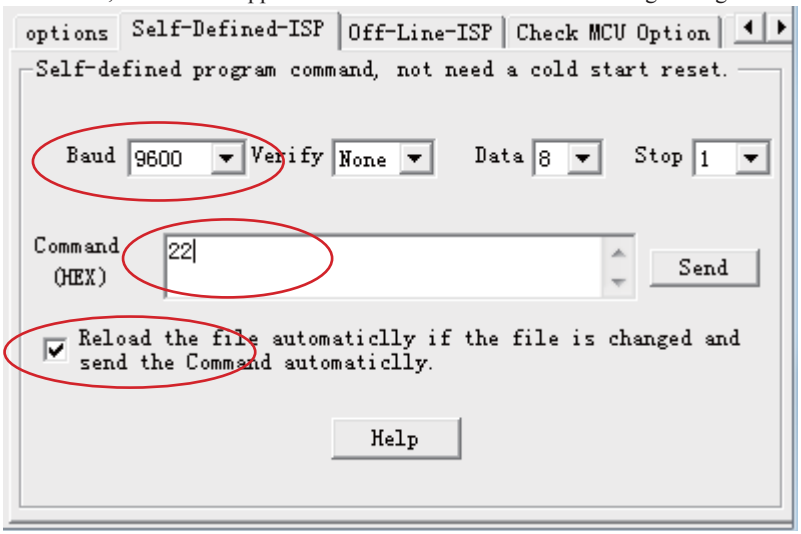
13.5 Self-Defined ISP download Demo

```
/*---------------------------------------------------------------------------------------------------------*/
/* --- STC MCU International Limited ---------------------------------------------------------------------*/
/* --- STC 1T Series MCU using software to custom download code Demo---------------------*/
/* If you want to use the program or the program referenced in the  ------------------------------*/
/* article, please specify in which data and procedures from STC    ------------------------------*/
/*---------------------------------------------------------------------------------------------------------*/
#include <reg51.h>
#include <instrins.h>

sfr IAP_CONTR = 0xe7;
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);
```

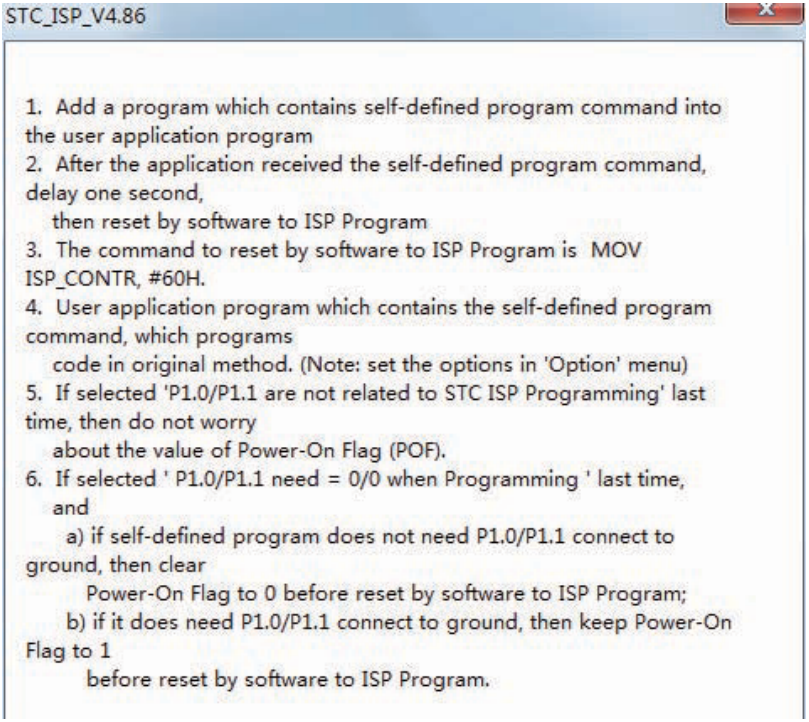
```
void main(void)
{
	unsigned char i = 0;

	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
	}
}
```

In addition, the PC-side application also need to make the following settings



Clicking the "Help" button as show in above figure, we can see the detail explaination as below.



Appendix A: Assembly Language Programming

INTRODUCTION

Assembly language is a computer language lying between the extremes of machine language and high-level language like Pascal or C use words and statements that are easily understood by humans, although still a long way from "natural" language.Machine language is the binary language of computers.A machine language program is a series of binary bytes representing instructions the computer can execute.

Assembly language replaces the binary codes of machine language with easy to remember "mnemonics"that facilitate programming.For example, an addition instruction in machine language might be represented by the code "10110011".It might be represented in assembly language by the mnemonic "ADD".Programming with mnemonics is obviously preferable to programming with binary codes.

Of course, this is not the whole story. Instructions operate on data, and the location of the data is specified by various "addressing modes" emmbeded in the binary code of the machine language instruction. So, there may be several variations of the ADD instruction, depending on what is added. The rules for specifying these variations are central to the theme of assembly language programming.

An assembly language program is not executable by a computer. Once written, the program must undergo translation to machine language. In the example above, the mnemonic "ADD" must be translated to the binary code "10110011". Depending on the complexity of the programming environment, this translation may involve one or more steps before an executable machine language program results. As a minimum, a program called an "assembler" is required to translate the instruction mnemonics to machine language binary codes. Afurther step may require a "linker" to combine portions of program from separate files and to set the address in memory at which th program may execute. We begin with a few definitions.

An assembly language program i a program written using labels, mnemonics, and so on, in which each statement corresponds to a machine instruction. Assembly language programs, often called source code or symbolic code, cannot be executed by a computer.

A machine language program is a program containing binary codes that represent instructions to a computer. Machine language programs, often called object code, are executable by a computer.

A assembler is a program that translate an assembly language program into a machine language program. The machine language program (object code) may be in "absolute" form or in "relocatable" form. In the latter case, "linking" is required to set the absolute address for execution.

A linker is a program that combines relocatable object programs (modules) and produces an absolute object program that is executable by a computer. A linker is sometimes called a "linker/locator" to reflect its separate functions of combining relocatable modules (linking) and setting the address for execution (locating).

A segment is a unit of code or data memory. A segment may be relocatable or absolute. A relocatable segment has a name, type, and other attributes that allow the linker to combine it with other paritial segments, if required, and to correctly locate the segment. An absolute segment has no name and cannot be combined with other segments.

A module contains one or more segments or partial segments. A module has a name assigned by the user. The module definitions determine the scope of local symbols. An object file contains one or more modules. A module may be thought of as a "file" in many instances.

A program consists of a single absolute module, merging all absolute and relocatable segments from all input modules. A program contains only the binary codes for instructions (with address and data constants) that are understood by a computer.

ASSEMBLER OPERATION

There are many assembler programs and other support programs available to facilitate the development of applications for the 8051 microcontroller. Intel's original MCS-51 family assembler, ASM51, is no longer available commercially. However, it set the standard to which the others are compared.

ASM51 is a powerful assembler with all the bells and whistles. It is available on Intel development systems and on the IBM PC family of microcomputers. Since these "host" computers contain a CPU chip other than the 8051, ASM51 is called a cross assembler. An 8051 source program may be written on the host computer (using any text editor) and may be assembled to an object file and listing file (using ASM51), but the program may not be executed. Since the host system's CPU chip is not an 8051, it does not understand the binary instruction in the object file. Execution on the host computer requires either hardware emulation or software simulation of the target CPU. A third possibility is to download the object program to an 8051-based target system for execution.

ASM51 is invoked from the system prompt by

ASM51 source_file [assembler_controls]

The source file is assembled and any assembler controls specified take effect. The assembler receives a source file as input (e.g., PROGRAM.SRC) and generates an object file (PROGRAM.OBJ) and listing file (PROGRAM.LST) as output. This is illustrated in Figure 1.

Since most assemblers scan the source program twice in performing the translation to machine language, they are described as two-pass assemblers. The assembler uses a location counter as the address of instructions and the values for labels. The action of each pass is described below.

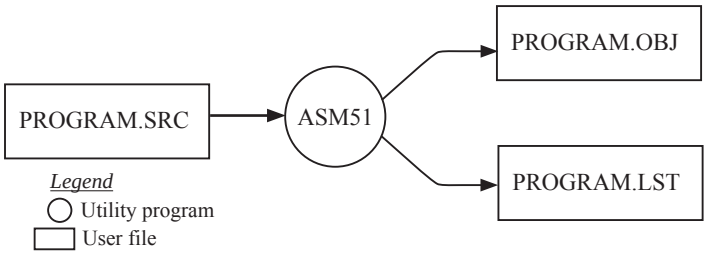


Figure 1 Assembling a source program

Pass one

During the first pass, the source file is scanned line-by-line and a symbol table is built. The location counter defaults to 0 or is set by the ORG (set origin) directive. As the file is scanned, the location counter is incremented by the length of each instruction. Define data directives (DBs or DWs) increment the location counter by the number of bytes defined. Reserve memory directives (DSs) increment the location counter by the number of bytes reserved.

Each time a label is found at the beginning of a line, it is placed in the symbol table along with the current value of the location counter. Symbols that are defined using equate directives (EQUs) are placed in the symbol table along with the "equated" value. The symbol table is saved and then used during pass two.

Pass two

During pass two, the object and listing files are created. Mnemonics are converted to opcodes and placed in the output files. Operands are evaluated and placed after the instruction opcodes. Where symbols appear in the operand field, their values are retrieved from the symbol table (created during pass one) and used in calculating the correct data or addresses for the instructions.

Since two passes are performed, the source program may use "forward references", that is, use a symbol before it is defined. This would occur, for example, in branching ahead in a program.

The object file, if it is absolute, contains only the binary bytes (00H-0FH) of the machine language program. A relocatable object file will also contain a sysmbol table and other information required for linking and locating. The listing file contains ASCII text codes (02H-7EH) for both the source program and the hexadecimal bytes in the machine language program.

A good demonstration of the distinction between an object file and a listing file is to display each on the host computer's CRT display (using, for example, the TYPE command on MS-DOS systems). The listing file clearly displays, with each line of output containing an address, opcode, and perhaps data, followed by the program statement from the source file. The listing file displays properly because it contains only ASCII text codes. Displaying the object file is a problem, however. The output will appear as "garbage", since the object file contains binary codes of an 8051 machine language program, rather than ASCII text codes.

ASSEMBLY LANGUAGE PROGRAM FORMAT

Assembly language programs contain the following:

- Machine instructions
- Assembler directives
- Assembler controls
- Comments

Machine instructions are the familiar mnemonics of executable instructions (e.g., ANL). Assembler directives are instructions to the assembler program that define program structure, symbols, data, constants, and so on (e.g., ORG). Assembler controls set assembler modes and direct assembly flow (e.g., $TITLE). Comments enhance the readability of programs by explaining the purpose and operation of instruction sequences.

Those lines containing machine instructions or assembler directives must be written following specific rules understood by the assembler. Each line is divided into "fields" separated by space or tab characters. The general format for each line is as follows:

[label:] mnemonic [operand] [, operand] […] [;commernt]

Only the mnemonic field is mandatory. Many assemblers require the label field, if present, to begin on the left in column 1, and subsequent fields to be separated by space or tab charecters. With ASM51, the label field needn't begin in column 1 and the mnemonic field needn't be on the same line as the label field. The operand field must, however, begin on the same line as the mnemonic field. The fields are described below.

Label Field

A label represents the address of the instruction (or data) that follows. When branching to this instruction, this label is usded in the operand field of the branch or jump instruction (e.g., SJMP SKIP).

Whereas the term "label" always represents an address, the term "symbol" is more general. Labels are one type of symbol and are identified by the requirement that they must terminate with a colon(:). Symbols are assigned values or attributes, using directives such as EQU, SEGMENT, BIT, DATA, etc. Symbols may be addresses, data constants, names of segments, or other constructs conceived by the programmer. Symbols do not terminate with a colon. In the example below, PAR is a symbol and START is a label (which is a type of symbol).

```
PAR     EQU   500           ;"PAR" IS A SYMBOL WHICH
                            ;REPRESENTS THE VALUE 500
START:  MOV   A,#0FFH       ;"START" IS A LABEL WHICH
                            ;REPRESENTS THE ADDRESS OF
                            ;THE MOV INSTRUCTION
```

A symbol (or label) must begin with a letter, question mark, or underscore (_); must be followed by letters, digit, "?", or "_"; and can contain up to 31 characters. Symbols may use upper- or lowercase characters, but they are treated the same. Reserved words (mnemonics, operators, predefined symbols, and directives) may not be used.

Mnemonic Field

Intruction mnemonics or assembler directives go into mnemonic field, which follows the label field. Examples of instruction mnemonics are ADD, MOV, DIV, or INC. Examples of assembler directives are ORG, EQU, or DB.

Operand Field

The operand field follows the mnemonic field. This field contains the address or data used by the instruction. A label may be used to represent the address of the data, or a symbol may be used to represent a data constant. The possibilities for the operand field are largely dependent on the operation. Some operations have no operand (e.g., the RET instruction), while others allow for multiple operands separated by commas. Indeed, the possibilties for the operand field are numberous, and we shall elaborate on these at length. But first, the comment field.

Comment Field

Remarks to clarify the program go into comment field at the end of each line. Comments must begin with a semicolon (;). Each lines may be comment lines by beginning them with a semicolon. Subroutines and large sections of a program generally begin with a comment block—serveral lines of comments that explain the general properties of the section of software that follows.

Special Assembler Symbols

Special assembler symbols are used for the register-specific addressing modes. These include A, R0 through R7, DPTR, PC, C and AB. In addition, a dollar sign ($) can be used to refer to the current value of the location counter. Some examples follow.

```
        SETB    C
        INC     DPTR
        JNB     TI , $
```

The last instruction above makes effective use of ASM51's location counter to avoid using a label. It could also be written as

```
HERE:   JNB     TI , HERE
```

Indirect Address

For certain instructions, the operand field may specify a register that contains the address of the data. The commercial "at" sign (@) indicates address indirection and may only be used with R0, R1, the DPTR, or the PC, depending on the instruction. For example,

```
        ADD     A , @R0
        MOVC    A , @A+PC
```

The first instruction above retrieves a byte of data from internal RAM at the address specified in R0. The second instruction retrieves a byte of data from external code memory at the address formed by adding the contents of the accumulator to the program counter. Note that the value of the program counter, when the add takes place, is the address of the instruction following MOVC. For both instruction above, the value retrieved is placed into the accumulator.

Immediate Data

Instructions using immediate addressing provide data in the operand field that become part of the instruction. Immediate data are preceded with a pound sign (#). For example,

```
CONSTANT    EQU    100
            MOV    A , #0FEH
            ORL    40H , #CONSTANT
```

All immediate data operations (except MOV DPTR,#data) require eight bits of data. The immediate data are evaluated as a 16-bit constant, and then the low-byte is used. All bits in the high-byte must be the same (00H or FFH) or the error message "value will not fit in a byte" is generated. For example, the following instructions are syntactically correct:

```
MOV    A , #0FF00H
MOV    A , #00FFH
```

But the following two instructions generate error messages:

```
MOV    A , #0FE00H
MOV    A , #01FFH
```

If signed decimal notation is used, constants from -256 to +255 may also be used. For example, the following two instructions are equivalent (and syntactically correct):

```
MOV    A , #-256
MOV    A , #0FF00H
```

Both instructions above put 00H into accumulator A.

Data Address

Many instructions access memory locations using direct addressing and require an on-chip data memory address (00H to 7FH) or an SFR address (80H to 0FFH) in the operand field. Predefined symbols may be used for the SFR addresses. For example,

```
MOV    A , 45H
MOV    A , SBUF          ;SAME AS MOV A, 99H
```

Bit Address

One of the most powerful features of the 8051 is the ability to access individual bits without the need for masking operations on bytes. Instructions accessing bit-addressable locations must provide a bit address in internal data memory (00h to 7FH) or a bit address in the SFRs (80H to 0FFH).

There are three ways to specify a bit address in an instruction: (a) explicitly by giving the address, (b) using the dot operator between the byte address and the bit position, and (c) using a predefined assembler symbol. Some examples follow.

```
SETB   0E7H              ;EXPLICIT BIT ADDRESS
SETB   ACC.7             ;DOT OPERATOR (SAME AS ABOVE)
JNB    TI , $            ;"TI" IS A PRE-DEFINED SYMBOL
JNB    99H , $           ;(SAME AS ABOVE)
```

Code Address

A code address is used in the operand field for jump instructions, including relative jumps (SJMP and conditional jumps), absolute jumps and calls (ACALL, AJMP), and long jumps and calls (LJMP, LCALL).

The code address is usually given in the form of a label.

ASM51 will determine the correct code address and insert into the instruction the correct 8-bit signed offset, 11-bit page address, or 16-bit long address, as appropriate.

Generic Jumps and Calls

ASM51 allows programmers to use a generic JMP or CALL mnemonic. "JMP" can be used instead of SJMP, AJMP or LJMP; and "CALL" can be used instead of ACALL or LCALL. The assembler converts the generic mnemonic to a "real" instruction following a few simple rules. The generic mnemonic converts to the short form (for JMP only) if no forward references are used and the jump destination is within -128 locations, or to the absolute form if no forward references are used and the instruction following the JMP or CALL instruction is in the same 2K block as the destination instruction. If short or absolute forms cannot be used, the conversion is to the long form.

The conversion is not necessarily the best programming choice. For example, if branching ahead a few instrucions, the generic JMP will always convert to LJMP even though an SJMP is probably better. Consider the following assembled instructions sequence using three generic jumps.

```
LOC     OBJ       LINE     SOURCE
1234                1              ORG     1234H
1234    04          2      START:  INC     A
1235    80FD        3              JMP     START          ;ASSEMBLES AS SJMP
12FC                4              ORG     START + 200
12FC    4134        5              JMP     START          ;ASSEMBLES AS AJMP
12FE    021301      6              JMP     FINISH         ;ASSEMBLES AS LJMP
1301    04          7      FINISH: INC     A
                    8              END
```

The first jump (line 3) assembles as SJMP because the destination is before the jump (i.e., no forward reference) and the offset is less than -128. The ORG directive in line 4 creates a gap of 200 locations between the label START and the second jump, so the conversion on line 5 is to AJMP because the offset is too great for SJMP. Note also that the address following the second jump (12FEH) and the address of START (1234H) are within the same 2K page, which, for this instruction sequence, is bounded by 1000H and 17FFH. This criterion must be met for absolute addressing. The third jump assembles as LJMP because the destination (FINISH) is not yet defined when the jump is assembled (i.e., a forward reference is used). The reader can verify that the conversion is as stated by examining the object field for each jump instruction.

ASSEMBLE-TIME EXPRESSION EVALUATION

Values and constants in the operand field may be expressed three ways: (a) explicitly (e.g.,0EFH), (b) with a predefined symbol (e.g., ACC), or (c) with an expression (e.g.,2 + 3). The use of expressions provides a powerful technique for making assembly language programs more readable and more flexible. When an expression is used, the assembler calculates a value and inserts it into the instruction.

All expression calculations are performed using 16-bit arithmetic; however, either 8 or 16 bits are inserted into the instruction as needed. For example, the following two instructions are the same:

```
MOV     DPTR,   #04FFH + 3
MOV     DPTR,   #0502H          ;ENTIRE 16-BIT RESULT USED
```

If the same expression is used in a "MOV A,#data" instruction, however, the error message "value will not fit in a byte" is generated by ASM51. An overview of the rules for evaluateing expressions follows.

Number Bases

The base for numeric constants is indicated in the usual way for Intel microprocessors. Constants must be followed with "B" for binary, "O" or "Q" for octal, "D" or nothing for decimal, or "H" for hexadecimal. For example, the following instructions are the same:

```
MOV     A , #15H
MOV     A , #1111B
MOV     A , #0FH
MOV     A , #17Q
MOV     A , #15D
```

Note that a digit must be the first character for hexadecimal constants in order to differentiate them from labels (i.e., "0A5H" not "A5H").

Charater Strings

Strings using one or two characters may be used as operands in expressions. The ASCII codes are converted to the binary equivalent by the assembler. Character constants are enclosed in single quotes ('). Some examples follow.

```
CJNE    A ,  # 'Q', AGAIN
SUBB    A ,  # '0'                ;CONVERT ASCII DIGIT TO BINARY DIGIT
MOV     DPTR, # 'AB'
MOV     DPTR, #4142H               ;SAME AS ABOVE
```

Arithmetic Operators

The arithmetic operators are

+	addition
-	subtraction
*	multiplication
/	division
MOD	modulo (remainder after division)

For example, the following two instructions are same:

```
MOV     A, 10 +10H
MOV     A, #1AH
```

The following two instructions are also the same:

```
MOV     A, #25 MOD 7
MOV     A, #4
```

Since the MOD operator could be confused with a symbol, it must be seperated from its operands by at least one space or tab character, or the operands must be enclosed in parentheses. The same applies for the other operators composed of letters.

Logical Operators

The logical operators are

OR	logical	OR
AND	logical	AND
XOR	logical	Exclusive OR
NOT	logical	NOT (complement)

The operation is applied on the corresponding bits in each operand. The operator must be separated from the operands by space or tab characters. For example, the following two instructions are the same:

```
MOV     A, # '9'  AND  0FH
MOV     A, #9
```

The NOT operator only takes one operand. The following three MOV instructions are the same:

```
THREE           EQU     3
MINUS_THREE     EQU     -3
                MOV     A, # (NOT THREE) + 1
                MOV     A, #MINUS_THREE
                MOV     A, #11111101B
```

Special Operators

The sepcial operators are

SHR	shift right
SHL	shift left
HIGH	high-byte
LOW	low-byte
()	evaluate first

For example, the following two instructions are the same:

```
MOV     A, #8  SHL  1
MOV     A, #10H
```

The following two instructions are also the same:

```
MOV     A, #HIGH  1234H
MOV     A, #12H
```

Relational Operators

When a relational operator is used between two operands, the result is alwalys false (0000H) or true (FFFFH). The operators are

EQ	=	equals
NE	< >	not equals
LT	<	less than
LE	<=	less than or equal to
GT	>	greater than
GE	>=	greater than or equal to

Note that for each operator, two forms are acceptable (e.g., "EQ" or "="). In the following examples, all relational tests are "true":

```
MOV     A, #5 = 5
MOV     A,#5 NE 4
MOV     A,# 'X'  LT  'Z'
MOV     A,# 'X'  >=  'X'
MOV     A,#$ > 0
MOV     A,#100 GE 50
```

So, the assembled instructions are equal to

```
MOV     A, #0FFH
```

Even though expressions evaluate to 16-bit results (i.e., 0FFFFH), in the examples above only the low-order eight bits are used, since the instruction is a move byte operation. The result is not considered too big in this case, because as signed numbers the 16-bit value FFFFH and the 8-bit value FFH are the same (-1).

Expression Examples

The following are examples of expressions and the values that result:

Expression	Result
'B' - 'A'	0001H
8/3	0002H
155 MOD 2	0001H
4 * 4	0010H
8 AND 7	0000H
NOT 1	FFFEH
'A' SHL 8	4100H
LOW 65535	00FFH
(8 + 1) * 2	0012H
5 EQ 4	0000H
'A' LT 'B'	FFFFH
3 <= 3	FFFFHss

A practical example that illustrates a common operation for timer initialization follows: Put -500 into Timer 1 registers TH1 and TL1. In using the HIGH and LOW operators, a good approach is

```
VALUE           EQU     -500
                MOV     TH1, #HIGH  VALUE
                MOV     TL1, #LOW VALUE
```

The assembler converts -500 to the corresponding 16-bit value (FE0CH); then the HIGH and LOW operators extract the high (FEH) and low (0CH) bytes. as appropriate for each MOV instruction.

Operator Precedence

The precedence of expression operators from highest to lowest is

```
( )
HIGH LOW
* / MOD SHL SHR
+ -
EQ NE LT LE GT GE = <> < <= > >=
NOT
AND
OR XOR
```

When operators of the same precedence are used, they are evaluated left to right.
Examples:

Expression	Value
HIGH ('A' SHL 8)	0041H
HIGH 'A' SHL 8	0000H
NOT 'A' - 1	FFBFH
'A' OR 'A' SHL 8	4141H

ASSEMBLER DIRECTIVES

Assembler directives are instructions to the assembler program. They are not assembly language instructions executable by the target microprocessor. However, they are placed in the mnemonic field of the program. With the exception of DB and DW, they have no direct effect on the contents of memory.

ASM51 provides several catagories of directives:

- Assembler state control (ORG, END, USING)
- Symbol definition (SEGMENT, EQU, SET, DATA, IDATA, XDATA, BIT, CODE)
- Storage initialization/reservation (DS, DBIT, DB, DW)
- Program linkage (PUBLIC, EXTRN,NAME)
- Segment selection (RSEG, CSEG, DSEG, ISEG, ESEG, XSEG)

Each assembler directive is presented below, ordered by catagory.

Assembler State Control

ORG (Set Origin) The format for the ORG (set origin) directive is

```
ORG     expression
```

The ORG directive alters the location counter to set a new program origin for statements that follow. A label is not permitted. Two examples follow.

```
ORG     100H                    ;SET LOCATION COUNTER TO 100H
ORG     ($ + 1000H) AND 0F00H   ;SET TO NEXT 4K BOUNDARY
```

The ORG directive can be used in any segment type. If the current segment is absolute, the value will be an absolute address in the current segment. If a relocatable segment is active, the value of the ORG expression is treated as an offset from the base address of the current instance of the segment.

End The format of the END directive is

```
END
```

END should be the last statement in the source file. No label is permitted and nothing beyond the END statement is processed by the assembler.

Using The format of the END directive is

```
USING   expression
```

This directive informs ASM51 of the currently active register bank. Subsequent uses of the predefined symbolic register addresses AR0 to AR7 will convert to the appropriate direct address for the active register bank. Consider the following sequence:

```
USING   3
PUSH    AR7
USING   1
PUSH    AR7
```

The first push above assembles to PUSH 1FH (R7 in bank 3), whereas the second push assembles to PUSH 0FH (R7 in bank 1).

Note that USING does not actually switch register banks; it only informs ASM51 of the active bank. Executing 8051 instructions is the only way to switch register banks. This is illustrated by modifying the example above as follows:

```
MOV     PSW, #00011000B         ;SELECT REGISTER BANK 3
USING   3
PUSH    AR7                     ;ASSEMBLE TO PUSH 1FH
MOV     PSW, #00001000B         ;SELECT REGISTER BANK 1
USING   1
PUSH    AR7                     ;ASSEMBLE TO PUSH 0FH
```

Symbol Definition

The symbol definition directives create symbols that represent segment, registers, numbers, and addresses. None of these directives may be preceded by a label. Symbols defined by these directives may not have been previously defined and may not be redefined by any means. The SET directive is the only exception. Symbol definiton directives are described below.

Segment The format for the SEGMENT directive is shown below.

```
symbol          SEGMENT         segment_type
```

The symbol is the name of a relocatable segment. In the use of segments, ASM51 is more complex than conventional assemblers, which generally support only "code" and "data" segment types. However, ASM51 defines additional segment types to accommodate the diverse memory spaces in the 8051. The following are the defined 8051 segment types (memory spaces):

- CODE (the code segment)
- XDATA (the external data space)
- DATA (the internal data space accessible by direct addressing, 00H–07H)
- IDATA (the entire internal data space accessible by indirect addressing, 00H–07H)
- BIT (the bit space; overlapping byte locations 20H–2FH of the internal data space)

For example, the statement

```
EPROM           SEGMENT         CODE
```

declares the symbol EPROM to be a SEGMENT of type CODE. Note that this statement simply declares what EPROM is. To actually begin using this segment, the RSEG directive is used (see below).

EQU (Equate) The format for the EQU directive is

```
Symbol          EQU     expression
```

The EQU directive assigns a numeric value to a specified symbol name. The symbol must be a valid symbol name, and the expression must conform to the rules described earlier.

The following are examples of the EQU directive:

```
N27             EQU     27              ;SET N27 TO THE VALUE 27
HERE            EQU     $               ;SET "HERE" TO THE VALUE OF
                                        ;THE LOCATION COUNTER
CR              EQU     0DH             ;SET CR (CARRIAGE RETURN) TO 0DH
MESSAGE:        DB 'This is a message'
LENGTH          EQU     $ - MESSAGE     ;"LENGTH" EQUALS LENGTH OF "MESSAGE"
```

Other Symbol Definition Directives The SET directive is similar to the EQU directive except the symbol may be redefined later, using another SET directive.

The DATA, IDATA, XDATA, BIT, and CODE directives assign addresses of the corresponding segment type to a symbol. These directives are not essential. A similar effect can be achieved using the EQU directive; if used, however, they evoke powerful type-checking by ASM51. Consider the following two directives and four instructions:

```
FLAG1    EQU    05H
FLAG2    BIT    05H
         SETB   FLAG1
         SETB   FLAG2
         MOV    FLAG1, #0
         MOV    FLAG2, #0
```

The use of FLAG2 in the last instruction in this sequence will generate a "data segment address expected" error message from ASM51. Since FLAG2 is defined as a bit address (using the BIT directive), it can be used in a set bit instruction, but it cannot be used in a move byte instruction. Hence, the error. Even though FLAG1 represents the same value (05H), it was defined using EQU and does not have an associated address space. This is not an advantage of EQU, but rather, a disadvantage. By properly defining address symbols for use in a specific memory space (using the directives BIT, DATA, XDATA,ect.), the programmer takes advantage of ASM51's powerful type-checking and avoids bugs from the misuse of symbols.

Storage Initialization/Reservation

The storage initialization and reservation directives initialize and reserve space in either word, byte, or bit units. The space reserved starts at the location indicated by the current value of the location counter in the currently active segment. These directives may be preceded by a label. The storage initialization/reservation directives are described below.

DS (Define Storage) The format for the DS (define storage) directive is

```
[label:]    DS    expression
```

The DS directive reserves space in byte units. It can be used in any segment type except BIT. The expression must be a valid assemble-time expression with no forward references and no relocatable or external references. When a DS statement is encountered in a program, the location counter of the current segment is incremented by the value of the expression. The sum of the location counter and the specified expression should not exceed the limitations of the current address space.

The following statement create a 40-byte buffer in the internal data segment:

```
         DSEG   AT      30H      ;PUT IN DATA SEGMENT (ABSOLUTE, INTERNAL)
LENGTH   EQU    40
BUFFER:  DS     LENGRH           ;40 BYTES RESERVED
```

The label BUFFER represents the address of the first location of reserved memory. For this example, the buffer begins at address 30H because "AT 30H" is specified with DSEG. The buffer could be cleared using the following instruction sequence:

```
         MOV    R7, #LENGTH
         MOV    R0, #BUFFER
LOOP:    MOV    @R0, #0
         DJNZ   R7, LOOP
         (continue)
```

To create a 1000-byte buffer in external RAM starting at 4000H, the following directives could be used:

```
XSTART      EQU     4000H
XLENGTH     EQU     1000
            XSEG      AT    XSTART
XBUFFER:    DS  XLENGTH
```

This buffer could be cleared with the following instruction sequence:

```
            MOV     DPTR, #XBUFFER
LOOP:       CLR     A
            MOVX    @DPTR, A
            INC     DPTR
            MOV     A, DPL
            CJNE    A, #LOW (XBUFFER + XLENGTH + 1), LOOP
            MOV     A, DPH
            CJNE    A, #HIGH (XBUFFER + XLENGTH + 1), LOOP
            (continue)
```

This is an excellent example of a powerful use of ASM51's operators and assemble-time expressions. Since an instruction does not exist to compare the data pointer with an immediate value, the operation must be fabricated from available instructions. Two compares are required, one each for the high- and low-bytes of the DPTR. Furthermore, the compare-and-jump-if-not-equal instruction works only with the accumulator or a register, so the data pointer bytes must be moved into the accumulator before the CJNE instruction. The loop terminates only when the data pointer has reached XBUFFER + LENGTH + 1. (The "+1" is needed because the data pointer is incremented after the last MOVX instruction.)

DBIT The format for the DBIT (define bit) directive is,

```
[label:]        DBIT    expression
```

The DBIT directive reserves space in bit units. It can be used only in a BIT segment. The expression must be a valid assemble-time expression with no forward references. When the DBIT statement is encountered in a program, the location counter of the current (BIT) segment is incremented by the value of the expression. Note that in a BIT segment, the basic unit of the location counter is bits rather than bytes. The following directives creat three flags in a absolute bit segment:

```
            BSEG            ;BIT SEGMENT (ABSOLUTE)
KEFLAG:     DBIT    1       ;KEYBOARD STATUS
PRFLAG:     DBIT    1       ;PRINTER STATUS
DKFLAG:     DBIT    1       ;DISK STATUS
```

Since an address is not specified with BSEG in the example above, the address of the flags defined by DBIT could be determined (if one wishes to to so) by examining the symbol table in the .LST or .M51 files. If the definitions above were the first use of BSEG, then KBFLAG would be at bit address 00H (bit 0 of byte address 20H). If other bits were defined previously using BSEG, then the definitions above would follow the last bit defined.

DB (Define Byte) The format for the DB (define byte) directive is,

```
[label:]        DB      expression  [, expression]  […]
```

The DB directive initializes code memory with byte values. Since it is used to actually place data constants in code memory, a CODE segment must be active. The expression list is a series of one or more byte values (each of which may be an expression) separated by commas.

The DB directive permits character strings (enclosed in single quotes) longer than two characters as long as they are not part of an expression. Each character in the string is converted to the corresponding ASCII code. If a label is used, it is assigned the address of th first byte. For example, the following statements

```
        CSEG  AT       0100H
SQUARES:  DB    0, 1, 4, 9, 16, 25          ;SQUARES OF NUMBERS 0-5
MESSAGE:  DB    'Login:', 0                 ;NULL-TERMINATED CHARACTER STRING
```

When assembled, result in the following hexadecimal memory assignments for external code memory:

Address	Contents
0100	00
0101	01
0102	04
0103	09
0104	10
0105	19
0106	4C
0107	6F
0108	67
0109	69
010A	6E
010B	3A
010C	00

DW (Define Word) The format for the DW (define word) directive is

```
[label:]    DW    expression    [, expression]  […]
```

The DW directive is the same as the DB directive except two memory locations (16 bits) are assigned for each data item. For example, the statements

```
CSEG    AT      200H
DW      $, 'A', 1234H, 2, 'BC'
```

result in the following hexadecimal memory assignments:

Address	Contents
0200	02
0201	00
0202	00
0203	41
0204	12
0205	34
0206	00
0207	02
0208	42
0209	43

Program Linkage

Program linkage directives allow the separately assembled modules (files) to communicate by permitting intermodule references and the naming of modules. In the following discussion, a "module" can be considered a "file." (In fact, a module may encompass more than one file.)

Public The format for the PUBLIC (public symbol) directive is

```
PUBLIC      symbol  [, symbol]  […]
```

The PUBLIC directive allows the list of specified symbols to known and used outside the currently assembled module. A symbol declared PUBLIC must be defined in the current module. Declaring it PUBLIC allows it to be referenced in another module. For example,

```
PUBLIC  INCHAR, OUTCHR, INLINE, OUTSTR
```

Extrn The format for the EXTRN (external symbol) directive is

```
EXTRN       segment_type  (symbol  [, symbol]  […], …)
```

The EXTRN directive lists symbols to be referenced in the current module that are defined in other modules. The list of external symbols must have a segment type associated with each symbol in the list. (The segment types are CODE, XDATA, DATA, IDATA, BIT, and NUMBER. NUMBER is a type-less symbol defined by EQU.) The segment type indicates the way a symbol may be used. The information is important at link-time to ensure symbols are used properly in different modules.

The PUBLIC and EXTRN directives work together. Consider the two files, MAIN.SRC and MESSAGES.SRC. The subroutines HELLO and GOOD_BYE are defined in the module MESSAGES but are made available to other modules using the PUBLIC directive. The subroutines are called in the module MAIN even though they are not defined there. The EXTRN directive declares that these symbols are defined in another module.

MAIN.SRC:

```
            EXTRN       CODE (HELLO, GOOD_BYE)
            …
            CALL        HELLO
            …
            CALL        GOOD_BYE
            …
            END
```

MESSAGES.SRC:

```
            PUBLIC      HELLO, GOOD_BYE
            …
HELLO:      (begin subroutine)
            …
            RET
GOOD_BYE:   (begin subroutine)
            …
            RET
            …
            END
```

Neither MAIN.SRC nor MESSAGES.SRC is a complete program; they must be assembled separately and linked together to form an executable program. During linking, the external references are resolved with correct addresses inserted as the destination for the CALL instructions.

Name The format for the NAME directive is

```
NAME    module_name
```

All the usual rules for symbol names apply to module names. If a name is not provided, the module takes on the file name (without a drive or subdirectory specifier and without an extension). In the absence of any use of the NAME directive, a program will contain one module for each file. The concept of "modules," therefore, is somewhat cumbersome, at least for relatively small programming problems. Even programs of moderate size (encompassing, for example, several files complete with relocatable segments) needn't use the NAME directive and needn't pay any special attention to the concept of "modules." For this reason, it was mentioned in the definition that a module may be considered a "file," to simplify learning ASM51. However, for very large programs (several thousand lines of code, or more), it makes sense to partition the problem into modules, where, for example, each module may encompass several files containing routines having a common purpose.

Segment Selection Directives

When the assembler encounters a segment selection directive, it diverts the following code or data into the selected segment until another segment is selected by a segment selection directive. The directive may select may select a previously defined relocatable segment or optionally create and select absolute segments.

RSEG (Relocatable Segment) The format for the RSEG (relocatable segment) directive is

```
RSEG        segment_name
```

Where "segment_name" is the name of a relocatable segment previously defined with the SEGMENT directive. RSEG is a "segment selection" directive that diverts subsequent code or data into the named segment until another segment selection directive is encountered.

Selecting Absolute Segments RSEG selects a relocatable segment. An "absolute" segment, on the other hand, is selected using one of the directives:

```
CSEG    (AT  address)
DSEG    (AT  address)
ISEG    (AT  address)
BSEG    (AT  address)
XSEG    (AT  address)
```

These directives select an absolute segment within the code, internal data, indirect internal data, bit, or external data address spaces, respectively. If an absolute address is provided (by indicating "AT address"), the assembler terminates the last absolute address segment, if any, of the specified segment type and creates a new absolute segment starting at that address. If an absolute address is not specified, the last absolute segment of the specified type is 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

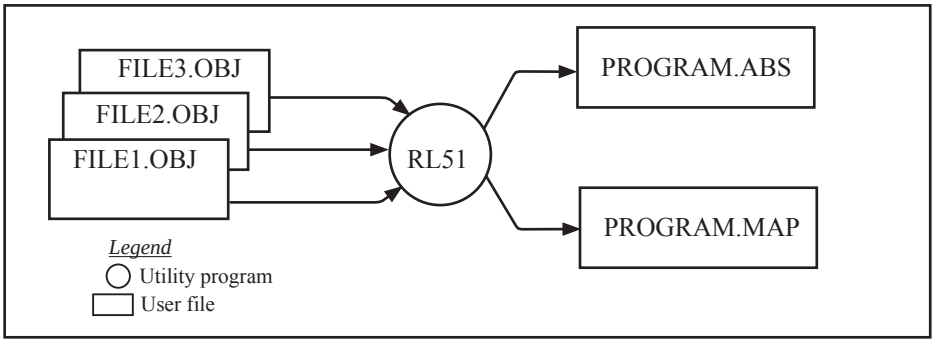
Assembler controls establish the format of the listing and object files by regulating the actions of ASM51. For the most part, assembler controls affect the look of the listing file, without having any affect on the program itself. They can be entered on the invocation line when a program is assembled, or they can be placed in the source file. Assembler controls appearing in the source file must be preceded with a dollor sign and must begin in column 1.

There are two categories of assembler controls: primary and general. Primary controls can be placed in the invocation line or at the beginnig of the source program. Only other primary controls may precede a primary control. General controls may be placed anywhere in the source program.

LINKER OPERATION

In developing large application programs, it is common to divide tasks into subprograms or modules containing sections of code (usually subroutines) that can be written separately from the overall program. The term "modular programming" refers to this programming strategy. Generally, modules are relocatable, meaning they are not intended for a specific address in the code or data space. A linking and locating program is needed to combine the modules into one absolute object module that can be executed.

Intel's RL51 is a typical linker/locator. It processes a series of relocatable object modules as input and creates an executable machine language program (PROGRAM, perhaps) and a listing file containing a memory map and symbol table (PROGRAM.M51). This is illustrated in following figure.



Linker operation

As relocatable modules are combined, all values for external symbols are resolved with values inserted into the output file. The linker is invoked from the system prompt by

RL51 input_list [T0 output_file] [location_controls]

The input_list is a list of relocatable object modules (files) separated by commas. The output_list is the name of the output absolute object module. If none is supplied, it defaults to the name of the first input file without any suffix. The location_controls set start addresses for the named segments.

For example, suppose three modules or files (MAIN.OBJ, MESSAGES.OBJ, and SUBROUTINES.OBJ) are to be combined into an executable program (EXAMPLE), and that these modules each contain two relocatable segments, one called EPROM of type CODE, and the other called ONCHIP of type DATA. Suppose further that the code segment is to be executable at address 4000H and the data segment is to reside starting at address 30H (in internal RAM). The following linker invocation could be used:

```
RS51 MAIN.OBJ, MESSAGES.OBJ, SUBROUTINES.OBJ TO EXAMPLE & CODE
(EPROM (4000H) DATA (ONCHIP (30H))
```

Note that the ampersand character "&" is used as the line continuaton character.

If the program begins at the label START, and this is the first instruction in the MAIN module, then execution begins at address 4000H. If the MAIN module was not linked first, or if the label START is not at the beginning of MAIN, then the program's entry point can be determined by examining the symbol table in the listing file EXAMPLE.M51 created by RL51. By default, EXAMPLE.M51 will contain only the link map. If a symbol table is desired, then each source program must have used the SDEBUG control. The following table shows the assembler controls supported by ASM51.

Assembler controls supported by ASM51				
NAME	PRIMARY/ GENERAL	DEFAULT	ABBREV.	MEANING
DATE (date)	P	DATE( )	DA	Place string in header (9 char. max.)
DEBUG	P	NODEBUG	DB	Outputs debug symbol information to object file
EJECT	G	not applicable	EJ	Continue listing on next page
ERRORPRINT (file)	P	NOERRORPRINT	EP	Designates a file to receive error messages in addition to the listing file (defauts to console)
NOERRORPRINT	P	NOERRORPRINT	NOEP	Designates that error messages will be printed in listing file only
GEN	G	GENONLY	GO	List only the fully expanded source as if all lines generated by a macro call were already in the source file
GENONLY	G	GENONLY	NOGE	List only the original source text in the listing file
INCLUED(file)	G	not applicable	IC	Designates a file to be included as part of the program
LIST	G	LIST	LI	Print subsequent lines of source code in listing file
NOLIST	G	LIST	NOLI	Do not print subsequent lines of source code in lisitng file
MACRO (men_precent)	P	MACRO(50)	MR	Evaluate and expand all macro calls. Allocate percentage of free memory for macro processing
NOMACRO	P	MACRO(50)	NOMR	Do not evalutate macro calls
MOD51	P	MOD51	MO	Recognize the 8051-specific predefined special function registers
NOMOD51	P	MOD51	NOMO	Do not recognize the 8051-specific predefined special function registers
OBJECT(file)	P	OBJECT(source.OBJ)	OJ	Designates file to receive object code
NOOBJECT	P	OBJECT(source.OBJ)	NOOJ	Designates that no object file will be created
PAGING	P	PAGING	PI	Designates that listing file be broken into pages and each will have a header
NOPAGING	P	PAGING	NOPI	Designates that listing file will contain no page breaks
PAGELENGTH (N)	P	PAGELENGT(60)	PL	Sets maximun number of lines in each page of listing file (range=10 to 65536)
PAGE WIDTH (N)	P	PAGEWIDTH(120)	PW	Set maximum number of characters in each line of listing file (range = 72 to 132)
PRINT(file)	P	PRINT(source.LST)	PR	Designates file to receive source listing
NOPRINT	P	PRINT(source.LST)	NOPR	Designates that no listing file will be created
SAVE	G	not applicable	SA	Stores current control settings from SAVE stack
RESTORE	G	not applicable	RS	Restores control settings from SAVE stack
REGISTERBANK (rb,...)	P	REGISTERBANK(0)	RB	Indicates one or more banks used in program module
NOREGISTER-BANK	P	REGISTERBANK(0)	NORB	Indicates that no register banks are used
SYMBOLS	P	SYMBOLS	SB	Creates a formatted table of all symbols used in program
NOSYMBOLS	P	SYMBOLS	NOSB	Designates that no symbol table is created
TITLE(string)	G	TITLE( )	TT	Places a string in all subsequent page headers (max.60 characters)
WORKFILES (path)	P	same as source	WF	Designates alternate path for temporay workfiles
XREF	P	NOXREF	XR	Creates a cross reference listing of all symbols used in program
NOXREF	P	NOXREF	NOXR	Designates that no cross reference list is created

MACROS

The macro processing facility (MPL) of ASM51 is a "string replacement" facility. Macros allow frequently used sections of code be defined once using a simple mnemonic and used anywhere in the program by inserting the mnemonic. Programming using macros is a powerful extension of the techniques described thus far. Macros can be defined anywhere in a source program and subsequently used like any other instruction. The syntax for macro definition is

```
%*DEFINE        (call_pattern)        (macro_body)
```

Once defined, the call pattern is like a mnemonic; it may be used like any assembly language instruction by placing it in the mnemonic field of a program. Macros are made distinct from "real" instructions by preceding them with a percent sign, "%". When the source program is assembled, everything within the macro-body, on a character-by-character basis, is substituted for the call-pattern. The mystique of macros is largely unfounded. They provide a simple means for replacing cumbersome instruction patterns with primitive, easy-to-remember mnemonics. The substitution, we reiterate, is on a character-by-character basis—nothing more, nothing less.

For example, if the following macro definition appears at the beginning of a source file,

```
%*DEFINE        (PUSH_DPTR)
                        (PUSH   DPH
                         PUSH   DPL
                         )
```

then the statement

```
%PUSH_DPTR
```

will appear in the .LST file as

```
PUSH    DPH
PUSH    DPL
```

The example above is a typical macro. Since the 8051 stack instructions operate only on direct addresses, pushing the data pointer requires two PUSH instructions. A similar macro can be created to POP the data pointer.

There are several distinct advantages in using macros:

- A source program using macros is more readable, since the macro mnemonic is generally more indicative of the intended operation than the equivalent assembler instructions.
- The source program is shorter and requires less typing.
- Using macros reduces bugs
- Using macros frees the programmer from dealing with low-level details.

The last two points above are related. Once a macro is written and debugged, it is used freely without the worry of bugs. In the PUSH_DPTR example above, if PUSH and POP instructions are used rather than push and pop macros, the programmer may inadvertently reverse the order of the pushes or pops. (Was it the high-byte or low-byte that was pushed first?) This would create a bug. Using macros, however, the details are worked out once—when the macro is written—and the macro is used freely thereafter, without the worry of bugs.

Since the replacement is on a character-by-character basis, the macro definition should be carefully constructed with carriage returns, tabs, ect., to ensure proper alignment of the macro statements with the rest of the assembly language program. Some trial and error is required.

There are advanced features of ASM51's macro-processing facility that allow for parameter passing, local labels, repeat operations, assembly flow control, and so on. These are discussed below.

Parameter Passing

A macro with parameters passed from the main program has the following modified format:

```
%*DEFINE        (macro_name  (parameter_list))    (macro_body)
```

For example, if the following macro is defined,

```
%*DEFINE        (CMPA#  (VALUE))
          (CJNE     A, #%VALUE, $ + 3
           )
```

then the macro call

```
%CMPA#  (20H)
```

will expand to the following instruction in the .LST file:

```
CJNE      A, #20H, $ + 3
```

Although the 8051 does not have a "compare accumulator" instruction, one is easily created using the CJNE instruction with "$+3" (the next instruction) as the destination for the conditional jump. The CMPA# mnemonic may be easier to remember for many programmers. Besides, use of the macro unburdens the programmer from remembering notational details, such as "$+3."

Let's develop another example. It would be nice if the 8051 had instructions such as

```
JUMP    IF  ACCUMULATOR  GREATER  THAN  X
JUMP    IF  ACCUMULATOR  GREATER  THAN  OR  EQUAL  TO  X
JUMP    IF  ACCUMULATOR  LESS  THAN  X
JUMP    IF  ACCUMULATOR  LESS  THAN  OR  EQUAL  TO  X
```

but it does not. These operations can be created using CJNE followed by JC or JNC, but the details are tricky. Suppose, for example, it is desired to jump to the label GREATER_THAN if the accumulator contains an ASCII code greater than "Z" (5AH). The following instruction sequence would work:

```
CJNE      A, #5BH, $÷3
JNC       GREATER_THAN
```

The CJNE instruction subtracts 5BH (i.e., "Z" + 1) from the content of A and sets or clears the carry flag accordingly. CJNE leaves C=1 for accumulator values 00H up to and including 5AH. (Note: 5AH-5BH<0, therefore C=1; but 5BH-5BH=0, therefore C=0.) Jumping to GREATER_THAN on the condition "not carry" correctly jumps for accumulator values 5BH, 5CH, 5DH, and so on, up to FFH. Once details such as these are worked out, they can be simplified by inventing an appropriate mnemonic, defining a macro, and using the macro instead of the corresponding instruction sequence. Here's the definition for a "jump if greater than" macro:

```
%*DEFINE        (JGT (VALUE,  LABEL))
                    (CJNE     A, #%VALUE+1, $+3      ;JGT
                     JNC       %LABEL
                     )
```

To test if the accumulator contains an ASCII code greater than "Z," as just discussed,the macro would be called as

```
%JGT      ('Z', GREATER_THAN)
```

ASM51 would expand this into

```
CJNE      A, #5BH, $+3        ;JGT
JNC       GREATER_THAN
```

The JGT macro is an excellent example of a relevant and powerful use of macros. By using macros, the programmer benefits by using a meaningful mnemonic and avoiding messy and potentially bug-ridden details.

Local Labels

Local labels may be used within a macro using the following format:

```
%*DEFINE        (macro_name  [(parameter_list)])
                        [LOCAL   list_of_local_labels]    (macro_body)
```

For example, the following macro definition

```
%*DEFINE    (DEC_DPTR)    LOCAL  SKIP
                (DEC     DPL                ;DECREMENT DATA POINTER
                 MOV     A, DPL
                 CJNE    A, #0FFH, %SKIP
                 DEC     DPL
%SKIP:           )
```

would be called as

```
%DEC_DPTR
```

and would be expanded by ASM51 into

```
            DEC     DPL                ;DECREMENT  DATA  POINTER
            MOV     A, DPL
            CJNE    A, #0FFH, SKIP00
            DEC     DPH
SKIP00:
```

Note that a local label generally will not conflict with the same label used elsewhere in the source program, since ASM51 appends a numeric code to the local label when the macro is expanded. Furthermore, the next use of the same local label receives the next numeric code, and so on.

The macro above has a potential "side effect." The accumulator is used as a temporary holding place for DPL. If the macro is used within a section of code that uses A for another purpose, the value in A would be lost. This side effect probably represents a bug in the program. The macro definition could guard against this by saving A on the stack. Here's an alternate definition for the DEC_DPTR macro:

```
%*DEFINE        (DEC_DPTR)      LOCAL SKIP
                  (PUSHACC
                 DEC     DPL                ;DECREMENT DATA POINTER
                 MOV     A, DPL
                 CJNE    A, #0FFH, %SKIP
                 DEC     DPH
%SKIP:           POP     ACC
                 )
```

Repeat Operations

This is one of several built-in (predefined) macros. The format is

```
%REPEAT         (expression)         (text)
```

For example, to fill a block of memory with 100 NOP instructions,

```
%REPEAT    (100)
 (NOP
 )
```

Control Flow Operations

The conditional assembly of section of code is provided by ASM51's control flow macro definition. The format is

```
%IF  (expression)   THEN  (balanced_text)
[ELSE     (balanced_text)]  FI
```

For example,

```
INTRENAL        EQU       1          ;1 = 8051 SERIAL I/O DRIVERS
                                     ;0 = 8251 SERIAL I/O DRIVERS
                .
                .
                %IF (INTERNAL) THEN
(INCHAR:        .                    ;8051 DRIVERS
                .
OUTCHR:         .
                .
                ) ELSE
(INCHAR:        .                    ;8251 DRIVERS
                .
OUTCHR:         .
                .
                )
```

In this example, the symbol INTERNAL is given the value 1 to select I/O subroutines for the 8051's serial port, or the value 0 to select I/O subroutines for an external UART, in this case the 8251. The IF macro causes ASM51 to assemble one set of drivers and skip over the other. Elsewhere in the program, the INCHAR and OUTCHR subroutines are used without consideration for the particular hardware configuration. As long as the program as assembled with the correct value for INTERNAL, the correct subroutine is executed.

Appendix B: 8051 C Programming

ADVANTAGES AND DISADVANTAGES OF 8051 C

The advantages of programming the 8051 in C as compared to assembly are:

- Offers all the benefits of high-level, structured programming languages such as C, including the ease of writing subroutines
- Often relieves the programmer of the hardware details that the complier handles on behalf of the programmer
- Easier to write, especially for large and complex programs
- Produces more readable program source codes

Nevertheless, 8051 C, being very similar to the conventional C language, also suffers from the following disadvantages:

- Processes the disadvantages of high-level, structured programming languages.
- Generally generates larger machine codes
- Programmer has less control and less ability to directly interact with hardware

To compare between 8051 C and assembly language, consider the solutions to the Example—Write a program using Timer 0 to create a 1KHz square wave on P1.0.
A solution written below in 8051 C language:

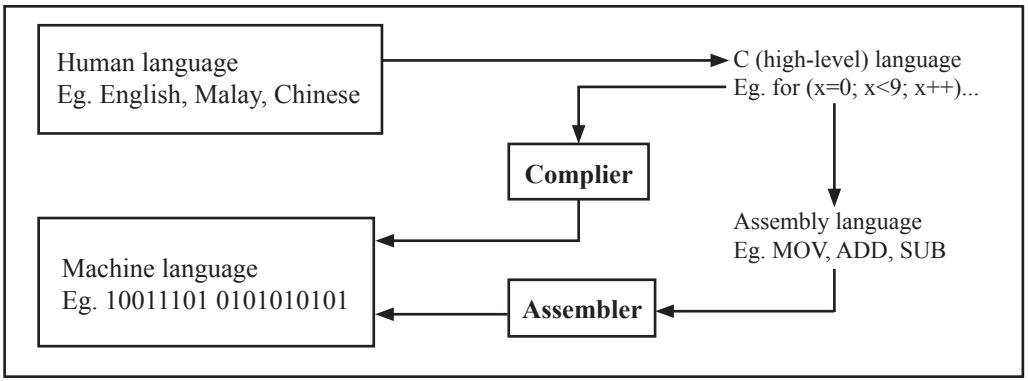
```
sbit portbit = P1^0;              /*Use variable portbit to refer to P1.0*/
main ( )
{
TMOD = 1;
while (1)
        {
        TH0 = 0xFE;
        TL0 = 0xC;
        TR0 = 1;
        while (TF0 !=1);
        TR0 = 0;
        TF0 = 0;
        portbit = !(P1.^0);
        }
}
```

A solution written below in assembly language:

```
            ORG     8100H
            MOV     TMOD, #01H      ;16-bit timer mode
LOOP:       MOV     TH0, #0FEH      ;-500 (high byte)
            MOV     TL0, #0CH       ;-500 (low byte)
            SETB    TR0             ;start timer
WAIT:       JNB     TF0, WAIT       ;wait for overflow
            CLR     TR0             ;stop timer
            CLR     TF0             ;clear timer overflow flag
            CPL     P1.0            ;toggle port bit
            SJMP    LOOP            ;repeat
            END
```

Notice that both the assembly and C language solutions for the above example require almost the same number of lines. However, the difference lies in the readability of these programs. The C version seems more human than assembly, and is hence more readable. This often helps facilitate the human programmer's efforts to write even very complex programs. The assembly language version is more closely related to the machine code, and though less readable, often results in more compact machine code. As with this example, the resultant machine code from the assembly version takes 83 bytes while that of the C version requires 149 bytes, an increase of 79.5%!

The human programmer's choice of either high-level C language or assembly language for talking to the 8051, whose language is machine language, presents an interesting picture, as shown in following figure.



Conversion between human, high-level, assembly, and machine language

8051 C COMPILERS

We saw in the above figure that a complier is needed to convert programs written in 8051 C language into machine language, just as an assembler is needed in the case of programs written in assembly language. A complier basically acts just like an assembler, except that it is more complex since the difference between C and machine language is far greater than that between assembly and machine language. Hence the complier faces a greater task to bridge that difference.

Currently, there exist various 8051 C complier, which offer almost similar functions. All our examples and programs have been compiled and tested with Keil's µ Vision 2 IDE by Keil Software, an integrated 8051 program development envrionment that includes its C51 cross compiler for C. A cross compiler is a compiler that normally runs on a platform such as IBM compatible PCs but is meant to compile programs into codes to be run on other platforms such as the 8051.

DATA TYPES

8051 C is very much like the conventional C language, except that several extensions and adaptations have been made to make it suitable for the 8051 programming environment. The first concern for the 8051 C programmer is the data types. Recall that a data type is something we use to store data. Readers will be familiar with the basic C data types such as int, char, and float, which are used to create variables to store integers, characters, or floating-points. In 8051 C, all the basic C data types are supported, plus a few additional data types meant to be used specifically with the 8051.

The following table gives a list of the common data types used in 8051 C. The ones in bold are the specific 8051 extensions. The data type **bit** can be used to declare variables that reside in the 8051's bit-addressable locations (namely byte locations 20H to 2FH or bit locations 00H to 7FH). Obviously, these bit variables can only store bit values of either 0 or 1. As an example, the following C statement:

```
bit   flag = 0;
```

declares a bit variable called flag and initializes it to 0.

Data types used in 8051 C language

Data Type	Bits	Bytes	Value Range
bit	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2,147,483,648 to +2,147,483,647
unsigned long	32	4	0 to 4,294,967,295
float	32	4	±1.175494E-38 to ±3.402823E+38
sbit	1		0 to 1
sfr	8	1	0 to 255
sfr16	16	2	0 to 65535

The data type **sbit** is somewhat similar to the bit data type, except that it is normally used to declare 1-bit variables that reside in special function registes (SFRs). For example:

```
sbit    P = 0xD0;
```

declares the **sbit** variable P and specifies that it refers to bit address D0H, which is really the LSB of the PSW SFR. Notice the difference here in the usage of the assignment ("=") operator. In the context of **sbit** declarations, it indicatess what address the **sbit** variable resides in, while in **bit** declarations, it is used to specify the initial value of the **bit** variable.

Besides directly assigning a bit address to an **sbit** variable, we could also use a previously defined **sfr** variable as the base address and assign our **sbit** variable to refer to a certain bit within that **sfr**. For example:

```
sfr     PSW = 0xD0;
sbit    P = PSW^0;
```

This declares an **sfr** variable called PSW that refers to the byte address D0H and then uses it as the base address to refer to its LSB (bit 0). This is then assigned to an **sbit** variable, P. For this purpose, the carat symbol (^) is used to specify bit position 0 of the PSW.

A third alternative uses a constant byte address as the base address within which a certain bit is referred. As an illustration, the previous two statements can be replaced with the following:

```
sbit    P = 0xD0 ^ 0;
```

Meanwhile, the **sfr** data type is used to declare byte (8-bit) variables that are associated with SFRs. The statement:

```
sfr     IE = 0xA8;
```

declares an **sfr** variable IE that resides at byte address A8H. Recall that this address is where the Interrupt Enable (IE) SFR is located; therefore, the sfr data type is just a means to enable us to assign names for SFRs so that it is easier to remember.

The **sfr16** data type is very similar to **sfr** but, while the **sfr** data type is used for 8-bit SFRs, **sfr16** is used for 16-bit SFRs. For example, the following statement:

```
sfr16   DPTR = 0x82;
```

declares a 16-bit variable DPTR whose lower-byte address is at 82H. Checking through the 8051 architecture, we find that this is the address of the DPL SFR, so again, the **sfr16** data type makes it easier for us to refer to the SFRs by name rather than address. There's just one thing left to mention. When declaring **sbit**, **sfr**, or **sfr16** variables, remember to do so outside main, otherwise you will get an error.

In actual fact though, all the SFRs in the 8051, including the individual flag, status, and control bits in the bit-addressable SFRs have already been declared in an include file, called reg51.h, which comes packaged with most 8051 C compilers. By using reg51.h, we can refer for instance to the interrupt enable register as simply IE rather than having to specify the address A8H, and to the data pointer as DPTR rather than 82H. All this makes 8051 C programs more human-readable and manageable. The contents of reg51.h are listed below.

```
/*-------------------------------------------------------------------------------------------------------------
REG51.H
Header file for generic 8051 microcontroller.
-------------------------------------------------------------------------------------------------------------*/
/* BYTE Register */
sfr     P0      = 0x80;
sfr     P1      = 0x90;
sfr     P2      = 0xA0;
sfr     P3      = 0xB0;
sfr     PSW     = 0xD0;
sfr     ACC     = 0xE0;
sfr     B       = 0xF0;
sfr     SP      = 0x81;
sfr     DPL     = 0x82;
sfr     DPH     = 0x83;
sfr     PCON    = 0x87;
sfr     TCON    = 0x88;
sfr     TMOD    = 0x89;
sfr     TL0     = 0x8A;
sfr     TL1     = 0x8B;
sfr     TH0     = 0x8C;
sfr     TH1     = 0x8D;
sfr     IE      = 0xA8;
sfr     IP      = 0xB8;
sfr     SCON    = 0x98;
sfr     SBUF    = 0x99;
/* BIT Register */
/* PSW */
sbit    CY      = 0xD7;
sbit    AC      = 0xD6;
sbit    F0      = 0xD5;
sbit    RS1     = 0xD4;
sbit    RS0     = 0xD3;
sbit    OV      = 0xD2;
sbit    P       = 0xD0;
/* TCON */
sbit    TF1     = 0x8F;
sbit    TR1     = 0x8E;
sbit    TF0     = 0x8D;
sbit    TR0     = 0x8C;
sbit    IE1     = 0x8B;
sbit    IT1     = 0x8A;
sbit    IE0     = 0x89;
sbit    IT0     = 0x88;
/* IE */
sbit    EA      = 0xAF;
sbit    ES      = 0xAC;
sbit    ET1     = 0xAB;
sbit    EX1     = 0xAA;
sbit    ET0     = 0xA9;
sbit    EX0     = 0xA8;
/* IP */
sbit    PS      = 0xBC;
sbit    PT1     = 0xBB;
sbit    PX1     = 0xBA;
sbit    PT0     = 0xB9;
sbit    PX0     = 0xB8;
/* P3 */
sbit    RD      = 0xB7;
sbit    WR      = 0xB6;
sbit    T1      = 0xB5;
sbit    T0      = 0xB4;
sbit    INT1    = 0xB3;
sbit    INT0    = 0xB2;
sbit    TXD     = 0xB1;
sbit    RXD     = 0xB0;
/* SCON */
sbit    SM0     = 0x9F;
sbit    SM1     = 0x9E;
sbit    SM2     = 0x9D;
sbit    REN     = 0x9C;
sbit    TB8     = 0x9B;
sbit    RB8     = 0x9A;
sbit    TI      = 0x99;
sbit    RI      = 0x98;
```

MEMORY TYPES AND MODELS

The 8051 has various types of memory space, including internal and external code and data memory. When declaring variables, it is hence reasonable to wonder in which type of memory those variables would reside. For this purpose, several memory type specifiers are available for use, as shown in following table.

Memory types used in 8051 C language	
Memory Type	Description (Size)
code	Code memory (64 Kbytes)
data	Directly addressable internal data memory (128 bytes)
idata	Indirectly addressable internal data memory (256 bytes)
bdata	Bit-addressable internal data memory (16 bytes)
xdata	External data memory (64 Kbytes)
pdata	Paged external data memory (256 bytes)

The first memory type specifier given in above table is **code**. This is used to specify that a variable is to reside in code memory, which has a range of up to 64 Kbytes. For example:

```
char        code        errormsg[ ] = "An error occurred" ;
```

declares a char array called errormsg that resides in code memory.

If you want to put a variable into data memory, then use either of the remaining five data memory specifiers in above table. Though the choice rests on you, bear in mind that each type of data memory affect the speed of access and the size of available data memory. For instance, consider the following declarations:

```
signed int    data    num1;
bit  bdata    numbit;
unsigned int  xdata   num2;
```

The first statement creates a signed int variable num1 that resides in inernal **data** memory (00H to 7FH). The next line declares a bit variable numbit that is to reside in the bit-addressable memory locations (byte addresses 20H to 2FH), also known as **bdata**. Finally, the last line declares an unsigned int variable called num2 that resides in external data memory, **xdata**. Having a variable located in the directly addressable internal data memory speeds up access considerably; hence, for programs that are time-critical, the variables should be of type **data**. For other variants such as 8052 with internal data memory up to 256 bytes, the **idata** specifier may be used. Note however that this is slower than data since it must use indirect addressing. Meanwhile, if you would rather have your variables reside in external memory, you have the choice of declaring them as **pdata** or **xdata**. A variable declared to be in **pdata** resides in the first 256 bytes (a page) of external memory, while if more storage is required, **xdata** should be used, which allows for accessing up to 64 Kbytes of external data memory.

What if when declaring a variable you forget to explicitly specify what type of memory it should reside in, or you wish that all variables are assigned a default memory type without having to specify them one by one? In this case, we make use of **memory models**. The following table lists the various memory models that you can use.

Memory models used in 8051 C language	
Memory Model	Description
Small	Variables default to the internal data memory (data)
Compact	Variables default to the first 256 bytes of external data memory (pdata)
Large	Variables default to external data memory (xdata)

A program is explicitly selected to be in a certain memory model by using the C directive, #pragma. Otherwise, the default memory model is **small**. It is recommended that programs use the small memory model as it allows for the fastest possible access by defaulting all variables to reside in internal data memory.

The **compact** memory model causes all variables to default to the first page of external data memory while the **large** memory model causes all variables to default to the full external data memory range of up to 64 Kbytes.

ARRAYS

Often, a group of variables used to store data of the same type need to be grouped together for better readability. For example, the ASCII table for decimal digits would be as shown below.

ASCII table for decimal digits	
Decimal Digit	ASCII Code In Hex
0	30H
1	31H
2	32H
3	33H
4	34H
5	35H
6	36H
7	37H
8	38H
9	39H

To store such a table in an 8051 C program, an array could be used. An array is a group of variables of the same data type, all of which could be accessed by using the name of the arrary along with an appropriate index.

The array to store the decimal ASCII table is:

```
int         table [10] =
{0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39};
```

Notice that all the elements of an array are separated by commas. To access an individul element, an index starting from 0 is used. For instance, table[0] refers to the first element while table[9] refers to the last element in this ASCII table.

STRUCTURES

Sometime it is also desired that variables of different data types but which are related to each other in some way be grouped together. For example, the name, age, and date of birth of a person would be stored in different types of variables, but all refer to the person's personal details. In such a case, a structure can be declared. A structure is a group of related variables that could be of different data types. Such a structure is declared by:

```
struct      person  {
                        char name;
                        int age;
                        long DOB;
                        };
```

Once such a structure has been declared, it can be used like a data type specifier to create structure variables that have the member's name, age, and DOB. For example:

```
struct      person      grace = {"Grace", 22, 01311980};
```

would create a structure variable grace to store the name, age, and data of birth of a person called Grace. Then in order to access the specific members within the person structure variable, use the variable name followed by the dot operator (.) and the member name. Therefore, grace.name, grace.age, grace.DOB would refer to Grace's name, age, and data of birth, respectively.

POINTERS

When programming the 8051 in assembly, sometimes register such as R0, R1, and DPTR are used to store the addresses of some data in a certain memory location. When data is accessed via these registers, indirect addressing is used. In this case, we say that R0, R1, or DPTR are used to point to the data, so they are essentially pointers.

Correspondingly in C, indirect access of data can be done through specially defined pointer variables. Pointers are simply just special types of variables, but whereas normal variables are used to directly store data, pointer variables are used to store the addresses of the data. Just bear in mind that whether you use normal variables or pointer variables, you still get to access the data in the end. It is just whether you go directly to where it is stored and get the data, as in the case of normal variables, or first consult a directory to check the location of that data before going there to get it, as in the case of pointer variables.
Declaring a pointer follows the format:

```
data_type     *pointer_name;
```
where

data_type	refers to which type of data that the pointer is pointing to
*	denotes that this is a pointer variable
pointer_name	is the name of the pointer

As an example, the following declarations:

```
int   * numPtr
int   num;
numPtr = &num;
```

first declares a pointer variable called numPtr that will be used to point to data of type int. The second declaration declares a normal variable and is put there for comparison. The third line assigns the address of the num variable to the numPtr pointer. The address of any variable can be obtained by using the address operator, &, as is used in this example. Bear in mind that once assigned, the numPtr pointer contains the address of the num variable, not the value of its data.

The above example could also be rewritten such that the pointer is straightaway initialized with an address when it is first declared:

```
int   num;
int   * numPtr = &num;
```

In order to further illustrate the difference between normal variables and pointer variables, consider the following, which is not a full C program but simply a fragment to illustrate our point:

```
int   num = 7;
int   * numPtr = &num;
printf ("%d\n", num);
printf ("%d\n", numPtr);
printf ("%d\n", &num);
printf ("%d\n", *numPtr);
```

The first line declare a normal variable, num, which is initialized to contain the data 7. Next, a pointer variable, numPtr, is declared, which is initialized to point to the address of num. The next four lines use the printf( ) function, which causes some data to be printed to some display terminal connected to the serial port. The first such line displays the contents of the num variable, which is in this case the value 7. The next displays the contents of the numPtr pointer, which is really some weird-looking number that is the address of the num variable. The third such line also displays the addresss of the num variable because the address operator is used to obtain num's address. The last line displays the actual data to which the numPtr pointer is pointing, which is 7. The * symbol is called the indirection operator, and when used with a pointer, indirectly obtains the data whose address is pointed to by the pointer. Therefore, the output display on the terminal would show:

```
7
13452  (or some other weird-looking number)
13452  (or some other weird-looking number)
7
```

A Pointer's Memory Type

Recall that pointers are also variables, so the question arises where they should be stored. When declaring pointers, we can specify different types of memory areas that these pointers should be in, for example:

```
int   * xdata numPtr = & num;
```

This is the same as our previous pointer examples. We declare a pointer numPtr, which points to data of type int stored in the num variable. The difference here is the use of the memory type specifier **xdata** after the *. This is specifies that pointer numPtr should reside in external data memory (**xdata**), and we say that the pointer's memory type is **xdata**.

Typed Pointers

We can go even further when declaring pointers. Consider the example:

```
int   data  * xdata   numPtr = &num;
```

The above statement declares the same pointer numPtr to reside in external data memory (**xdata**), and this pointer points to data of type int that is itself stored in the variable num in internal data memory (**data**). The memory type specifier, **data**, before the * specifies the *data memory type* while the memory type specifier, **xdata**, after the * specifies the pointer memory type.

Pointer declarations where the data memory types are explicitly specified are called typed pointers. Typed pointers have the property that you specify in your code where the data pointed by pointers should reside. The size of typed pointers depends on the data memory type and could be one or two bytes.

Untyped Pointers

When we do not explicitly state the data memory type when declaring pointers, we get untyped pointers, which are generic pointers that can point to data residing in any type of memory. Untyped pointers have the advantage that they can be used to point to any data independent of the type of memory in which the data is stored. All untyped pointers consist of 3 bytes, and are hence larger than typed pointers. Untyped pointers are also generally slower because the data memory type is not determined or known until the complied program is run at runtime. The first byte of untyped pointers refers to the data memory type, which is simply a number according to the following table. The second and third bytes are,respectively,the higher-order and lower-order bytes of the address being pointed to.

An untyped pointer is declared just like normal C, where:

```
int   * xdata  numPtr = &num;
```

does not explicitly specify the memory type of the data pointed to by the pointer. In this case, we are using untyped pointers.

Data memory type values stored in first byte of untyped pointers	
Value	Data Memory Type
1	idata
2	xdata
3	pdata
4	data/bdata
5	code

FUNCTIONS

In programming the 8051 in assembly, we learnt the advantages of using subroutines to group together common and frequently used instructions. The same concept appears in 8051 C, but instead of calling them subroutines, we call them **functions**. As in conventional C, a function must be declared and defined. A function definition includes a list of the number and types of inputs, and the type of the output (return type), puls a description of the internal contents, or what is to be done within that function.

The format of a typical function definition is as follows:

```
return_type    function_name (arguments)    [memory]  [reentrant]  [interrupt]  [using]
{
        …
}
```

where

return_type	refers to the data type of the return (output) value
function_name	is any name that you wish to call the function as
arguments	is the list of the type and number of input (argument) values
memory	refers to an explicit memory model (small, compact or large)
reentrant	refers to whether the function is reentrant (recursive)
interrupt	indicates that the function is acctually an ISR
using	explicitly specifies which register bank to use

Consider a typical example, a function to calculate the sum of two numbers:

```
int  sum (int a, int b)
{
        return  a + b;
}
```

This function is called sum and takes in two arguments, both of type int. The return type is also int, meaning that the output (return value) would be an int. Within the body of the function, delimited by braces, we see that the return value is basically the sum of the two agruments. In our example above, we omitted explicitly specifying the options: memory, reentrant, interrupt, and using. This means that the arguments passed to the function would be using the default small memory model, meaning that they would be stored in internal data memory. This function is also by default non-recursive and a normal function, not an ISR. Meanwhile, the default register bank is bank 0.

Parameter Passing

In 8051 C, parameters are passed to and from functions and used as function arguments (inputs). Nevertheless, the technical details of where and how these parameters are stored are transparent to the programmer, who does not need to worry about these techinalities. In 8051 C, parameters are passed through the register or through memory. Passing parameters through registers is faster and is the default way in which things are done. The registers used and their purpose are described in more detail below.

Registers used in parameter passing				
Number of Argument	Char / 1-Byte Pointer	INT / 2-Byte Pointer	Long/Float	Generic Pointer
1	R7	R6 & R7	R4–R7	R1–R3
2	R5	R4 &R5	R4–R7	
3	R3	R2 & R3		

Since there are only eight registers in the 8051, there may be situations where we do not have enough registers for parameter passing. When this happens, the remaining parameters can be passed through fixed memory loacations. 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

Appendix C: STC12C2052AD series Electrical Characteristics

Absolute Maximum Ratings

Parameter	Symbol	Min	Max	Unit
Srotage temperature	TST	-55	+125	℃
Operating temperature (I)	TA	-40	+85	℃
Operating temperature (C)	TA	0	+70	℃
DC power supply (5V)	VDD - VSS	-0.3	+6.0	V
DC power supply (3V)	VDD - VSS	-0.3	+4.0	V
Voltage on any pin	-	-0.5	5.5	V

DC Specification (5V MCU)

Sym	Parameter	Specification				Test Condition
		Min.	Typ	Max.	Unit	
V_{DD}	Operating Voltage	3.5	5.0	5.5	V	
I_{PD}	Power Down Current	-	< 0.1	-	uA	5V
I_{IDL}	Idle Current	-	3.0	-	mA	5V
I_{CC}	Operating Current	-	4	20	mA	5V
V_{IL1}	Input Low Voltage(P0,P1,P2,P3)	-	-	0.8	V	5V
V_{IL2}	Input Low Voltage (RESET, XTAL1)			1.5	V	5V
V_{IH1}	Input High Voltage (P0,P1,P2,P3)	2.0	-	-	V	5V
V_{IH2}	Input High Voltage (RESET)	3.0	-	-	V	5V
I_{OL1}	Sink Current for output low (P0,P1,P2,P3)	-	20	-	mA	5V
I_{OH1}	Sourcing Current for output high (P0,P1,P2,P3) (Quasi-output)	150	230	-	uA	5V
I_{OH2}	Sourcing Current for output high (P0,P1,P2,P3) (Push-Pull, Strong-output)	-	20	-	mA	5V
I_{IL}	Logic 0 input current (P0,P1,P2,P3)	-	18	50	uA	Vpin=0V
I_{TL}	Logic 1 to 0 transition current (P0,P1,P2,P3)	-	270	600	uA	Vpin=2.0V

DC Specification (3V MCU)

Sym	Parameter	Specification				Test Condition
		Min.	Typ	Max.	Unit	
V_{DD}	Operating Voltage	2.2	3.3	3.8	V	
I_{PD}	Power Down Current	-	<0.1	-	uA	3.3V
I_{IDL}	Idle Current	-	2.0	-	mA	3.3V
I_{CC}	Operating Current	-	4	10	mA	3.3V
V_{IL1}	Input Low (P0,P1,P2,P3)	-	-	0.8	V	3.3V
V_{IL2}	Input low voltage (RESET,XTAL1)	-	-	1.5	V	3.3V
V_{IH1}	Input High (P0,P1,P2,P3)	2.0	-	-	V	3.3V
V_{IH2}	Input High (RESET)	3.0	-	-	V	3.3V
I_{OL1}	Sink Current for output low (P0,P1,P2,P3)	-	20	-	mA	3.3V@Vpin=0.45V
I_{OH1}	Sourcing Current for output high (P0,P1,P2,P3) (Quasi-output)	40	70	-	uA	3.3V
I_{OH2}	Sourcing Current for output high (P0,P1,P2,P3) (Push-Pull)	-	20	-	mA	3.3V
I_{IL}	Logic 0 input current (P0,P1,P2,P3)	-	8	50	uA	Vpin=0V
I_{TL}	Logic 1 to 0 transition current (P0,P1,P2,P3)	-	110	600	uA	Vpin=2.0V

Appendix D: Program for indirect addressing inner 256B RAM

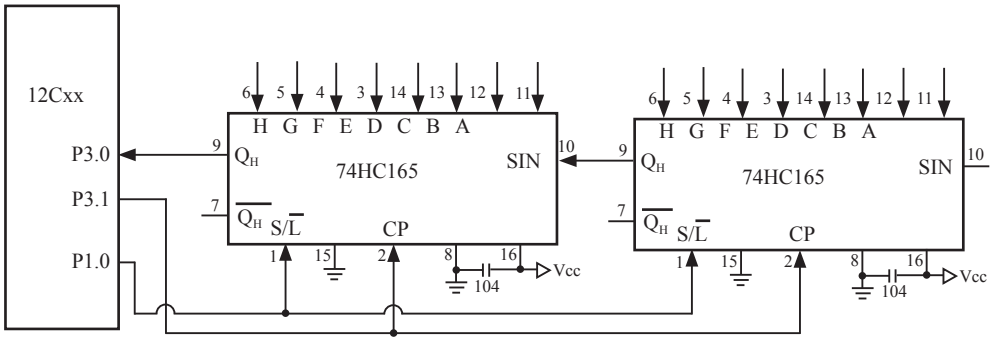
```
;/*---------------------------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------------------------------------*/
;/* --- STC 1T Series MCU  the inner 256B normal RAM (indirect addressing) Demo ----------*/
;/* If you want to use the program or the program referenced in the  ---------------------------------*/
;/* article, please specify in which data and procedures from STC    ---------------------------------*/
;/*---------------------------------------------------------------------------------------------------------*/

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

        ORG     0050H
INITIAL:
        MOV     R0,     #253

        MOV     R1,     #3H
TEST_ALL_RAM:
        MOV     R2,     #0FFH
TEST_ONE_RAM:
        MOV     A,      R2
        MOV     @R1,    A
        CLR     A
        MOV     A,      @R1

        CJNE    A,      2H,     ERROR_DISPLAY
        DJNZ    R2,     TEST_ONE_RAM
        INC     R1
        DJNZ    R0,     TEST_ALL_RAM
OK_DISPLAY:
        MOV     P1,     #11111110B
Wait1:
        SJMP    Wait1

ERROR_DISPLAY:
        MOV     A,      R1
        MOV     P1,     A
Wait2:
        SJMP    Wait2
        END
```

Appendix E: Using Serial port expand I/O interface

STC12C2052AD series MCU serial port mode0 can be used for expand IO if UART is free in your application. UART Mode0 is a synchronous shift register, the baudrate is fixed at fosc/12, RXD pin (P3.0) is the data I/O port, and TXD pin (P3.1) is clock output port, data width is 8 bits, always sent / received the lowest bit at first.

(1) Using 74HC165 expand parallel input ports

Please refer to the following circuit which using 2 pcs 74HC165 to expand 16 input I/Os

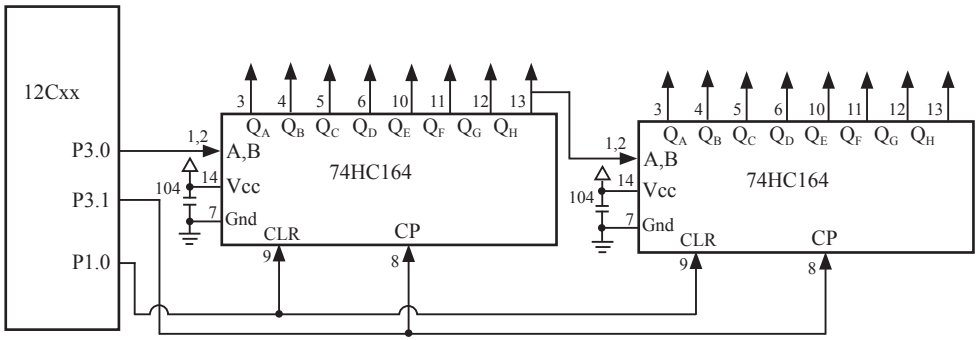


74HC165 is a 8-bit parallel input shift register, when S/L (Shift/Load) pin is falling to low level, the parallel port data is read into internal register, and now, if S/L is raising to high and ClockDisable pin (15 pin) is low level, then clock signal from CP pin is enable. At this time register data will be output from the Dh pin (9 pin) with the clock input.

```
        …
        MOV R7,#05H                 ;read 5 groups data
        MOV R0,#20H                 ;set buffer address
START:  CLR P1.0                    ;S/L = 0, load port data
        SETB P1.0                   ;S/L = 1, lock data and enable clock
        MOV R1,#02H                 ;2 bytes per group
RXDAT:MOV SCON,#00010000B           ;set serial as mode 0 and enable receive data
WAIT:   JNB RI,WAIT                 ;wait for receive complete
        CLR RI                      ;clear receive complete flag
        MOV A,SBUF                  ;read data to ACC
        MOV @R0,A                   ;save data to buffer
        INC R0                      ;modify buffer ptr
        DJNZ R1,RXDAT               ;read next byte
        DJNZ R7,START               ;read next group
        …
```

(2) Using 74HC164 expand parallel output ports

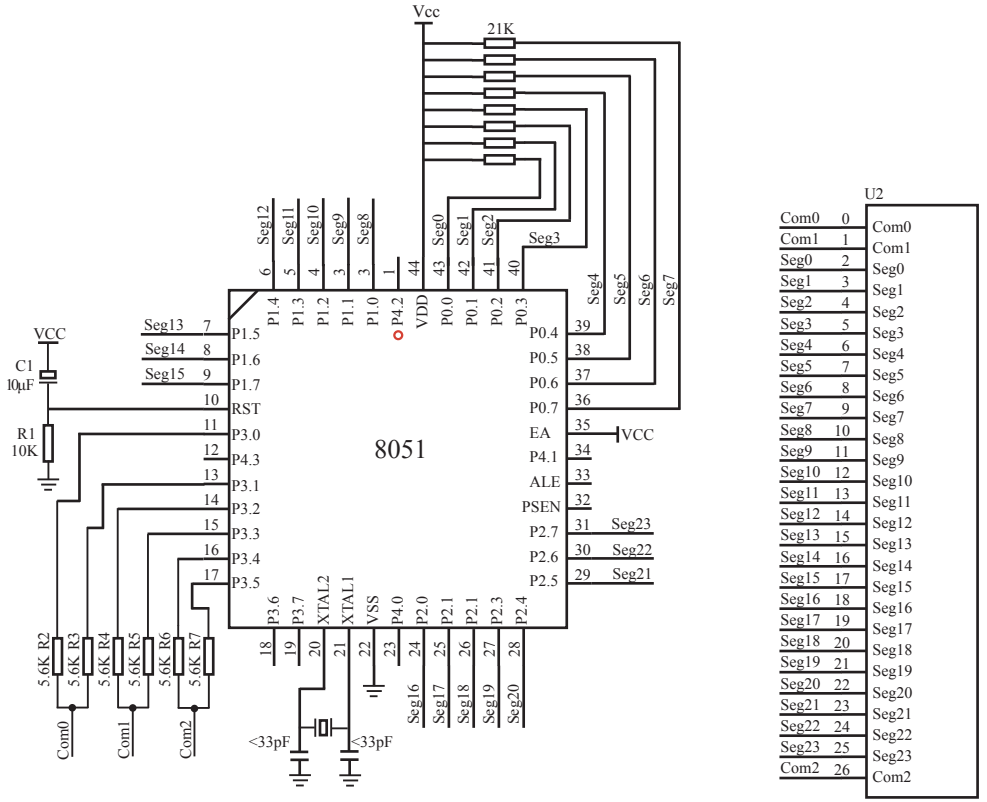
Please refer to the following circuit which using 2 pcs 74HC164 to expand 16 output I/Os



When serial port is working in MODE0, the serial data is input/output from RXD(P3.0) pin and serial clock is output from TXD(P3.1). Serial data is always starting transmission from the lowest bit.

```
        …
START:  MOV R7,#02H             ;output 2 bytes data
        MOV R0,#30H             ;set buffer address
        MOV SCON,#00000000B     ;set serial as mode 0
SEND:   MOV A,@R0               ;read data from buffer
        MOV SBUF,A              ;start send data
WAIT:   JNB TI,WAIT             ;wait for send complete
        CLR TI                  ;clear send complete flag
        INC R0                  ;modify buffer ptr
        DJNZ R7,SEND            ;send next data
        …
```

Appendix F: Use STC MCU common I/O driving LCD Display



```
NAME LcdDriver
#include<reg52.h>

;*******************************************************************************
;the LCD is 1/3 duty and 1/3 bias; 3Com*24Seg; 9 display RAM;
;
;                          Bit7    Bit6    Bit5    Bit4    Bit3    Bit2    Bit1    Bit0
;Com0:  Com0Data0:          Seg7    Seg6    Seg5    Seg4    Seg3    Seg2    Seg1    Seg0
;       Com0Data1:          Seg15   Seg14   Seg13   Seg12   Seg11   Seg10   Seg9    Seg8
;       Com0Data2:          Seg23   Seg22   Seg21   Seg20   Seg19   Seg18   Seg17   Seg16
;Com1:  Com1Data0:          Seg7    Seg6    Seg5    Seg4    Seg3    Seg2    Seg1    Seg0
;       Com1Data1:          Seg15   Seg14   Seg13   Seg12   Seg11   Seg10   Seg9    Seg8
;       Com1Data2:          Seg23   Seg22   Seg21   Seg20   Seg19   Seg18   Seg17   Seg16
;Com2:  Com2Data0:          Seg7    Seg6    Seg5    Seg4    Seg3    Seg2    Seg1    Seg0
;       Com2Data1:          Seg15   Seg14   Seg13   Seg12   Seg11   Seg10   Seg9    Seg8
;       Com2Data2:          Seg23   Seg22   Seg21   Seg20   Seg19   Seg18   Seg17   Seg16
;*******************************************************************************
;Com0:  P3^0,P3^1  when P3^0 = P3^1 = 1          then Com0=VCC(=5V);
;                       P3^0 = P3^1 = 0         then Com0=GND(=0V);
;                       P3^0 = 1, P3^1=0        then Com0=1/2 VCC;
;Com1:  P3^2,P3^3   the same as the Com0
;Com2:  P3^4,P3^5   the same as the Com0
;

sbit    SEG0  =P0^0
sbit    SEG1  =P0^1
sbit    SEG2  =P0^2
sbit    SEG3  =P0^3
sbit    SEG4  =P0^4
sbit    SEG5  =P0^5
sbit    SEG6  =P0^6
sbit    SEG7  =P0^7
sbit    SEG8  =P1^0
sbit    SEG9  =P1^1
sbit    SEG10 =P1^2
```

```
sbit	SEG11 =P1^3
sbit	SEG12 =P1^4
sbit	SEG13 =P1^5
sbit	SEG14 =P1^6
sbit	SEG15 =P1^7
sbit	SEG16 =P2^0
sbit	SEG17 =P2^1
sbit	SEG18 =P2^2
sbit	SEG19 =P2^3
sbit	SEG20 =P2^4
sbit	SEG21 =P2^5
sbit	SEG22 =P2^6
sbit	SEG23 =P2^7
;*************************************************************************************

;======Interrupt================================
	CSEG	AT	0000H
	LJMP	start

	CSEG	AT	000BH
	LJMP	int_t0

;======register================================
lcdd_bit SEGMENT BIT
	RSEG lcdd_bit
	OutFlag:	DBIT 1		;the output display reverse flag
lcdd_data SEGMENT DATA
	RSEG lcdd_data
	Com0Data0:	DS	1
	Com0Data1:	DS	1
	Com0Data2:	DS	1
	Com1Data0:	DS	1
	Com1Data1:	DS	1
	Com1Data2:	DS	1
	Com2Data0:	DS	1
	Com2Data1:	DS	1
	Com2Data2:	DS	1
	TimeS:		DS	1
```

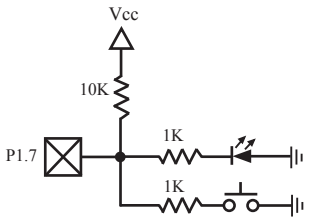
```
;======Interrupt Code============================
t0_int    SEGMENT         CODE
          RSEG    t0_int
          USING  1
;***************************************************************
;Time0 interrupt
;ths system crystalloid is 22.1184MHz
;the time to get the Time0 interrupr is 2.5mS
;the whole duty is 2.5mS*6=15mS, including reverse
;***************************************************************
int_t0:
          ORL     TL0,#00H
          MOV     TH0,#0EEH
          PUSH    ACC
          PUSH    PSW
          MOV     PSW,#08H
          ACALL OutData
          POP     PSW
          POP     ACC
          RETI

;======SUB CODE==================================
uart_sub SEGMENT CODE
          RSEG    uart_sub
          USING  0
;***************************************************************
;initial the display RAM data
;if want to display other,then you may add other data to this RAM
;Com0:   Com0Data0,Com0Data1,Com0Data2
;Com1:   Com1Data0,Com1Data1,Com1Data2
;Com2:   Com2Data0,Com0Data1,Com0Data2
;***************************************************************
InitComData:                ;it will display "11111111"
          MOV     Com0Data0,      #24H
          MOV     Com0Data1,      #49H
          MOV     Com0Data2,      #92H
```

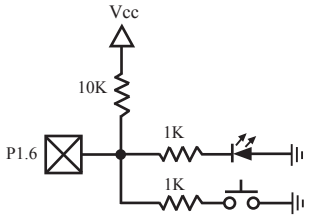
```
        MOV     Com1Data0,      #92H
        MOV     Com1Data1,      #24H
        MOV     Com1Data2,      #49H
        MOV     Com2Data0,      #00H
        MOV     Com2Data1,      #00H
        MOV     Com2Data2,      #00H
        RET

;*********************************************************************
;reverse the display data
;*********************************************************************
RetComData:
        MOV     R0,     #Com0Data0          ;get the first data address
        MOV     R7,     #9
RetCom_0:
        MOV     A,      @R0
        CPL     A
        MOV     @R0,    A
        INC     R0
        DJNZ    R7,     RetCom_0
        RET
;*********************************************************************
;get the display Data and send to Output register
;*********************************************************************
OutData:
        INC     TimeS
        MOV     A,      TimeS
        MOV     P3,     #11010101B              ;clear display,all Com are 1/2VCC and invalidate
        CJNE    A,      #01H,   OutData_1       ;judge the duty
        MOV     P0,     Com0Data0
        MOV     P1,     Com0Data1
        MOV     P2,     Com0Data2
        JNB     OutFlag,OutData_00
        MOV     P3,     #11010111B              ;Com0 is work and is VCC
        RET
```

```
OutData_00:
        MOV     P3,     #11010100B          ;Com0 is work and is GND
        RET
OutData_1:
        CJNE    A,      #02H,OutData_2
        MOV     P0,     Com1Data0
        MOV     P1,     Com1Data1
        MOV     P2,     Com1Data2
        JNB     OutFlag,OutData_10
        MOV     P3,     #11011101B          ;Com1 is work and is VCC
        RET
OutData_10:
        MOV     P3,     #11010001B          ;Com1 is work and is GND
        RET
OutData_2:
        MOV     P0,     Com2Data0
        MOV     P1,     Com2Data1
        MOV     P2,     Com2Data2
        JNB     OutFlag,OutData_20
        MOV     P3,     #11110101B          ;Com2 is work and is VCC
        SJMP    OutData_21
OutData_20:
        MOV  P3,#11000101B          ;Com2 is work and is GND
OutData_21:
        MOV     TimeS,  #00H
        ACALL RetComData
        CPL     OutFlag
        RET

;======Main Code================================
uart_main SEGMENT CODE
        RSEG    uart_main
        USING  0
```

```
start:
        MOV    SP,#40H
        CLR    OutFlag
        MOV    TimeS,#00H
        MOV    TL0,#00H
        MOV    TH0,#0EEH
        MOV    TMOD,#01H
        MOV    IE,#82H
        ACALL InitComData
        SETB   TR0
Main:
        NOP
        SJMP   Main

        END
```

Appendix G: LED driven by an I/O port and Key Scan





It can save a lot of I/O ports that STC12C2052AD MCU I/O ports can used as the LED drivers and key detection concurrently because of their feature which they can be set to the weak pull , the strong pull (push-pull) output, only input (high impedance), open drain four modes.

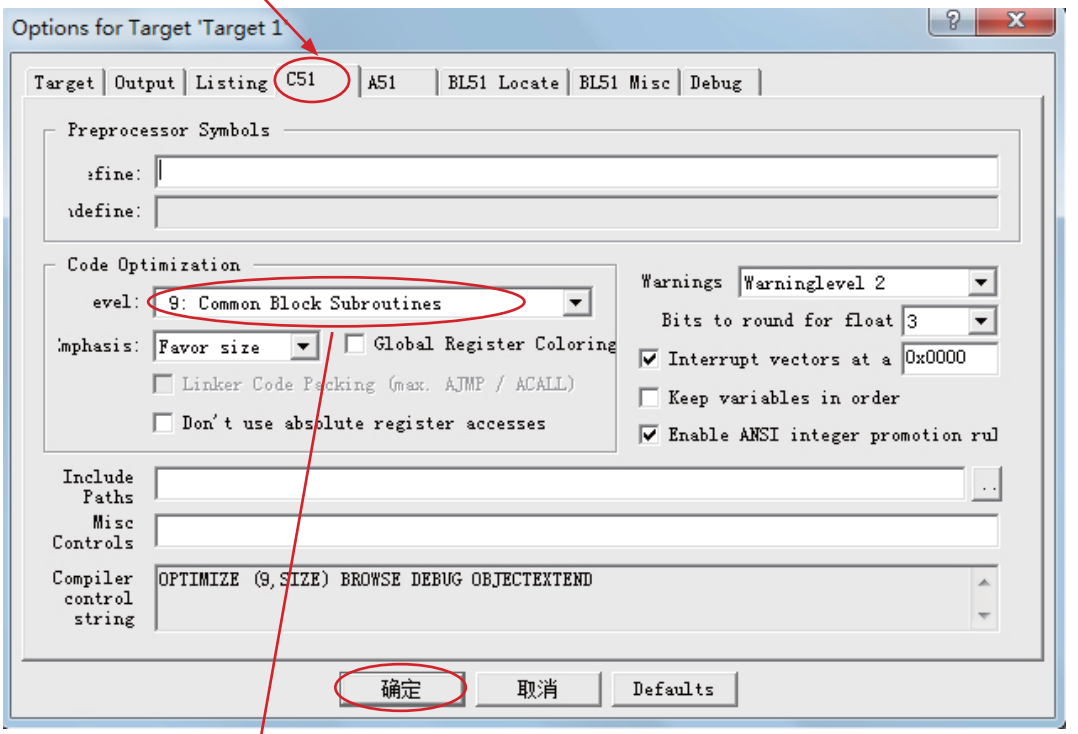
When driving the LED, the I/O port should be set as strongly push-pull output, and the LED will be lighted when the output is high.

When testing the keys, the I/O port should be set as weak pull input, and then reading the status of external ports can test the keys.

Appendix H: How to reduce the Length of Code through Keil C

Setting as shown below in Keil C can maximum reduce about 10K to the length of original code

1. Choose the "Options for Target" in "Project" menu
2. Choose the option "C51" in "Options for Target"



3. Code Optimization, 9 common block subroutines
4. Click "OK", compile the program once again.

Appendix I: Notes of STC12C2052AD series Application

About reset circuit

If the system frequency is below 12MHz, the external reset circuit is not required. Reset pin can be connected to ground through the 1K resistor or can be connected directly to ground. The proposal to create PCB to retain RC reset circuit

About Clock oscillator

If you need to use internal RC oscillator (4MHz ~ 8MHz because of manufacturing error and temperature drift), XTAL1 pin and XTAL2 pin must be floating. If you use a external active crystal oscillator, clock signal input from XTAL1 pin and XTAL2 pin floating.

About power

Power at both ends need to add a 47uF electrolytic capacitor and a 0.1uF capacitor, to remove the coupling and filtering.

Appendix J: Notes of STC12 series Replaced Traditional 8051

STC12C2052AD series MCU Timer0/Timer1/UART is fully compatible with the traditional 8051 MCU. After power on reset, the default input clock source is the divider 12 of system clock frequency, and UART baudrate generator is Timer 1. MCU instruction execution speed is faster than the traditional 8051 MCU 8 ~ 12 times in the same working environment,so software delay programs need to be adjusted.

ALE

Traditional 8051's ALE pin output signal on divide 6 the system clock frequency can be externally provided clock, if disable ALE output in STC12C2052AD series system, you can get clock source from CLKOUT0/ P1.0, CLKOUT1/P1.1 or XTAL2 clock output. (Recommended a 200ohm series resistor to the XTAL2 pin). ALE pin is an disturbance source when traditional 8051's system clock frequency is too high. STC89xx series MCU add ALEOFFF bit in AUXR register. While STC12C2052AD series MCU directly disable ALE pin dividing 6 the system clock output, and can remove ALE disturbance thoroughly. Please compare the following two registers.

AUXR register of STC89xx series

Mnemonic	Add	Name	Bit7	Bit6	Bit5	Bit4	Bir3	Bit2	Bit1	Bit0	Reset Value
AUXR	8EH	Auxiliary register 0	-	-	-	-	-	-	EXTRAM	ALEOFF	xxxx,xx00

AUXR register of STC12C2052AD series

Mnemonic	Add	Name	Bit7	Bit6	Bit5	Bit4	Bir3	Bit2	Bit1	Bit0	Reset Value
AUXR	8EH	Auxiliary register	T0x12	T1x12	UART_M0x6	EADCI	ESPI	ELVDI	-	-	000x,xxxx

PSEN

Traditional 8051 execute external program through the PSEN signal, STC12C2052AD series is system MCU concept, integrated high-capacity internal program memory, do not need external program memory expansion generally, so have no PSEN signal, PSEN pin can be used as GPIO.

General Qusi-Bidirectional I/O

Traditional 8051 access I/O (signal transition or read status) timing is 12 clocks, STC12C2052AD series MCU is 4 clocks. When you need to read an external signal, if internal output a rising edge signal, for the traditional 8051, this process is 12 clocks, you can read at once, but for STC12C2052AD series MCU, this process is 4 clocks, when internal instructions is complete but external signal is not ready, so you must delay 1~2 nop operation.

I/O drive capability

STC12C2052AD series I/O port sink drive current is 20mA, has a strong drive capability, the port is not burn out when drive high current generally. STC89 series I/O port sink drive current is only 6mA, is not enough to drive high current. For the high current drive applications, it is strongly recommended to use STC12C2052AD series MCU.

Power consumption

Power consumption consists of two parts: crystal oscillator amplifier circuits and digital circuits. For crystal oscillator amplifier circuits, STC12C2052AD series is lower then STC89 series. For digital circuits, the higher clock frequency, the greater the power consumption. STC12C2052AD series MCU instruction execution speed is faster than theSTC89 series MCU 3~24 times in the same working environment, so if you need to achieve the same efficiency, STC12C2052AD series required frequency is lower than STC89 series MCU.