STC89Cxx series MCU
STC89LExx series MCU
Data Sheet

CONTENTS

Chapter 1 Introduction

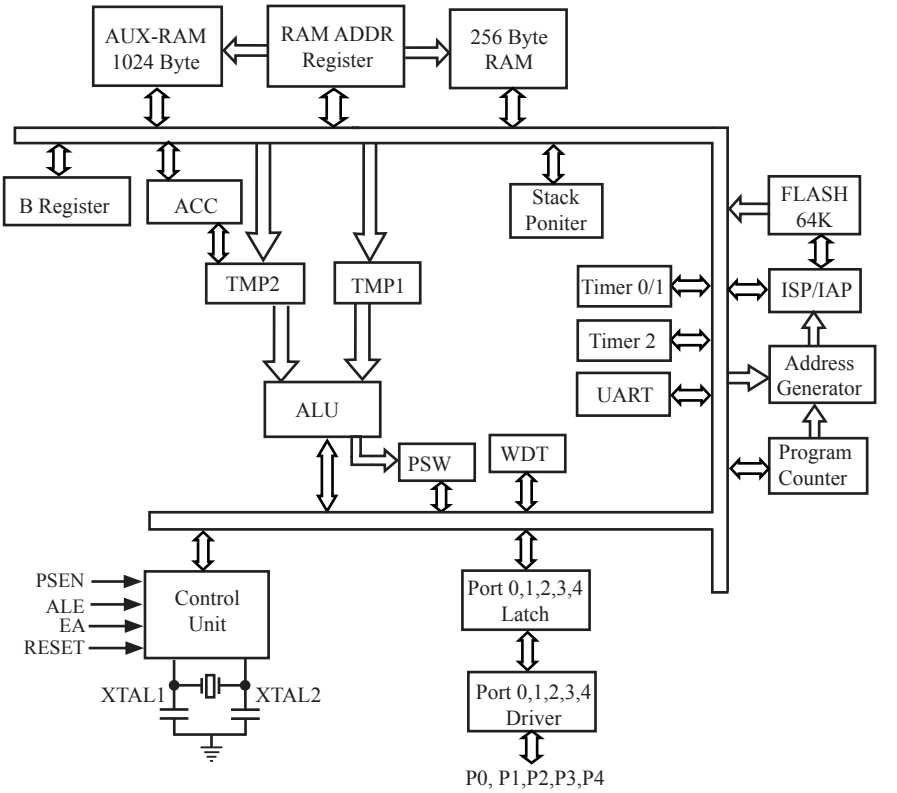
STC89xx series, which is produced by STC MCU Limited, is a 8-bit single-chip microcontroller with a fully compatible instruction set with industrial-standard 80C51 series microcontroller. There is 64K bytes flash memory embeded for appliaction program, which is shared with In-System-Programming code.In-System-Programming (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. There are 1280 bytes or 512 bytes on-chip RAM embedded that provides requirement from wide field application. The user can configure the device to run in 12 clocks per machine cycle, and to get the same performance just as he uses another standard 80C51 device that is provided by other vendor, or 6 clocks per machine cycle to achieve twice performance. The STC89xx series retain all features of the standard 80C51. In addition, the STC89xx series have a extra I/O port (P4), Timer 2, a 8-sources, 4-priority-level interrupt structure, on-chip crystal oscillator,and a one-time enabled Watchdog Timer.

1.1 Features

- Enhanced 80C51 Central Processing Unit ,6T or 12T per machine cycle
- Operation voltage range: 5.5V~3.3V (STC89C series) or 2.0V~ 3.6V (STC89LE series)
- Operation frequency range: 0- 48MHz@12T, or 0-24MHz@6T
- On-chip 4/8/13/16/20/32/64K FLASH program memory with flexible ISP/IAP capability
- On-chip 1280 byte / 512 byte RAM
- Be capable of addressing up to 64K byte of external RAM
- Be capable of addressing up to 64K bytes external memory
- Dual Data Pointer (DPTR) to speed up data movement
- Three 16-bit timer/counter, Timer 2 is an up/down counter with programmable clcok output on P1.0
- 8 vector-address, 4 level priority interrupt capability
- One enhanced UART with hardware address-recognition, frame-error detection function, and with self baud-rate generator.
- One 15 bits Watch-Dog-Timer with 8-bit pre-scaler (one-time-enabled)
- integrate MAX810 — specialized reset circuit
- Three power management modes: idle mode and power-down mode
- Low EMI: inhibit ALE emission
- Power down mode can be woken-up by INT0/P3.2 pin, INT1/P3.3 pin, T0/P3.4, T1/P3.5, RXD/P3.0 pin, INT2/P4.3, INT3/P4.2
- Maximum 39 programmable I/O ports are available
- Four 8-bit bi-directonal ports; extra four-bit additional P4 are available for PLCC-44 and LQFP-44
- Operating temperature: -40 ~ +85^{o}C (industrial) / 0~75^{o}C (commercial)
- package type :LQFP-44,PDIP-40,PLCC-44

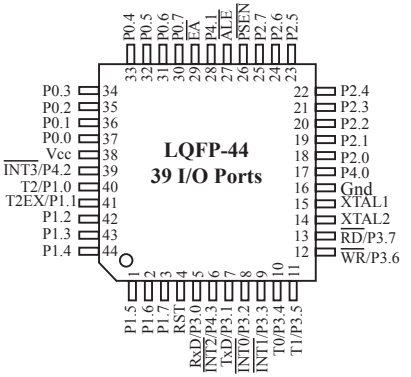
1.2 Block diagram

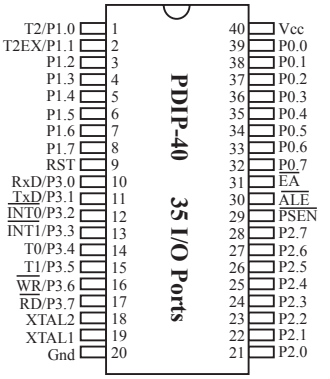
The CPU kernel of STC89xx series are fully compatible to the standard 8051 microcontroller, maintains all instruction mnemonics and binary compatibility. STC89xx series can execute the fastest instructions per 6 clock cycles or 12 clock cycles(as the same as the standard 80C51) . Improvement of individual programs depends on the actual instructions used.

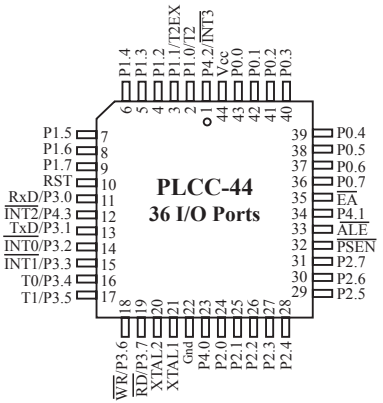


STC89xx Block Diagram

1.3 Pin Configurations





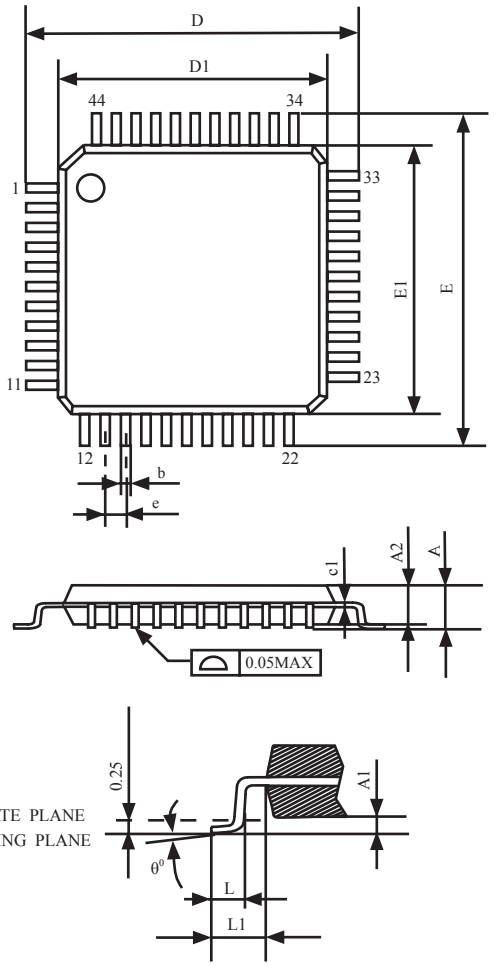


1.4 Pin Descriptions

MNEMONIC	LQFP44	PDIP40	PLCC44	DESCRIPTION
P0.0 ~ P0.7	37~30	39~32	43~36	Port0 :Port0 is an 8-bit bi-directional I/O port with pull-up resistance. Except being as GPIO, Port 0 is also the multiplexed low-order address and data bus during accesses to external program and data memory.
P1.0 ~ P1.7	40~44	1-8	2~9	Port1 : General-purposed I/O with weak pull-up resistance inside. When 1s are written into Port1, the strong output driving CMOS only turn-on two period and then the weak pull-up resistance keep the port high.
	1~3			
P1.0/T2	40	1	2	
P1.1/T2EX	41	2	3	P1.0 is also used as one of event sources for timer2, or output carrier of timer 2, alias T2. P1.1 is also used as one of interrupt-controlling sources for time 2, alias T2EX.
P2.0 ~ P2.7	18-25	21-28	24~31	Port2 : Port2 is an 8-bit bi-directional I/O port with pull-up resistance. Except being as GPIO, Port2 emits the high-order address byte during accessing to external program and data memory.
P3.0/RxD	5	10	11	Port3 : General-purposed I/O with weak pull-up resistance inside. When 1s are written into Port1, the strong output driving CMOS only turn-on two period and then the weak pull-up resistance keep the port high. Port3 also serves the functions of various special features. P3.0 and P3.1 act as receiver and transceiver of the data for UART function block, Alias RxD and TxD. P3.2 and P3.3 also act as external interrupt sources, alias /INT0 and /INT1. P3.4 and P3.5 also act as event sources for timer0 and timer1 individually, alias T0 and T1. P3.6 also acts as write signal while access to external memory,alias /WR. P3.7 also acts as read signal while access to external memory, alias /RD.
P3.1/TxD	7	11	13	
P3.2/INT0	8	12	14	
P3.3/INT1	9	13	15	
P3.4/T0	10	14	16	
P3.5/T1	11	15	17	
P3.6/WR	12	16	18	
P3.7/RD	13	17	19	
P4.0	17		23	Port4 : Port4 are extended I/O ports such like Port1. It can be available only on LQFP-44, PLCC-44. P4.2 and P4.3 also act as external interrupt sources, alias /INT3 and /INT2.
P4.1	28		34	
P4.2/INT3	39		1	
P4.3/INT2	6		12	
RST	4	9	10	RESET : A high on this pin for at least two machine cycles will reset the device.
/EA	29	31	35	EA must be kept at low to enable the device to fetch program code from external flash memory. An internal pull-up resistance has been embedded in this pin.
/ALE	27	30	33	Output pulse for latching the low byte of address during accesses to external memory.
/PSEN	26	29	32	The read strobe to external program memory, low active.
XTAL1	15	19	21	Crystal 1: Input to the inverting oscillator amplifier.Receives the external oscillator signal when an external oscillator is used.
XTAL2	14	18	20	Crystal 2: Output from the inverting amplifier. This pin should be floated when an external oscillator is used.
VCC	38	40	44	Power
Gnd	16	20	22	Ground

1.5 Pin Package Drawings

LQFP-44 OUTLINE PACKAGE

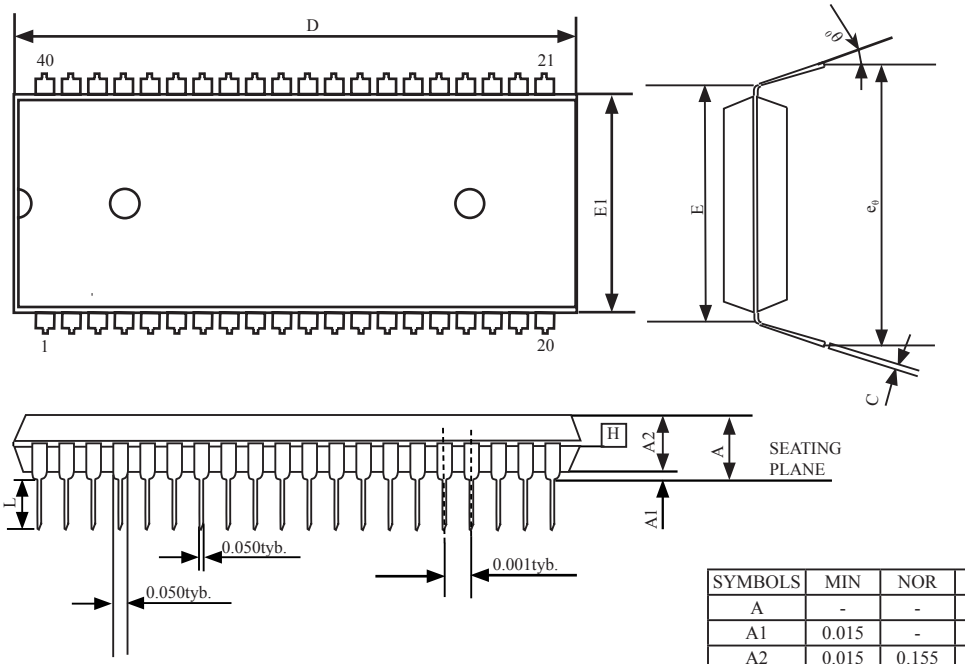


VARIATIONS (ALL DIMENSIONS SHOWN IN MM

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

NOTES:
1.JEDEC OUTLINE:MS-026 BSB
2.DIMENSIONS D1 AND E1 D0 NOT INCLUDE MOLD PROTRUSION. ALLOWBLE PROTRUSION IS 0.25mm PER SIDE. D1 AND E1 ARE MAXIMUM PLASTIC BODY SIZE DIMENSIONS IMCLUDING MOLD MISMATCH.
3.DIMENSION b DOES NOT INCLUDE DAMBAR PROTRUSION.ALLOWBLE DAMBAR PROTRUSION SHALL NOT CAUSE THE LEAD WIDTH TO EXCEED THE MAXIMUN b DIMNSION BY MORE THAN 0.08mm.

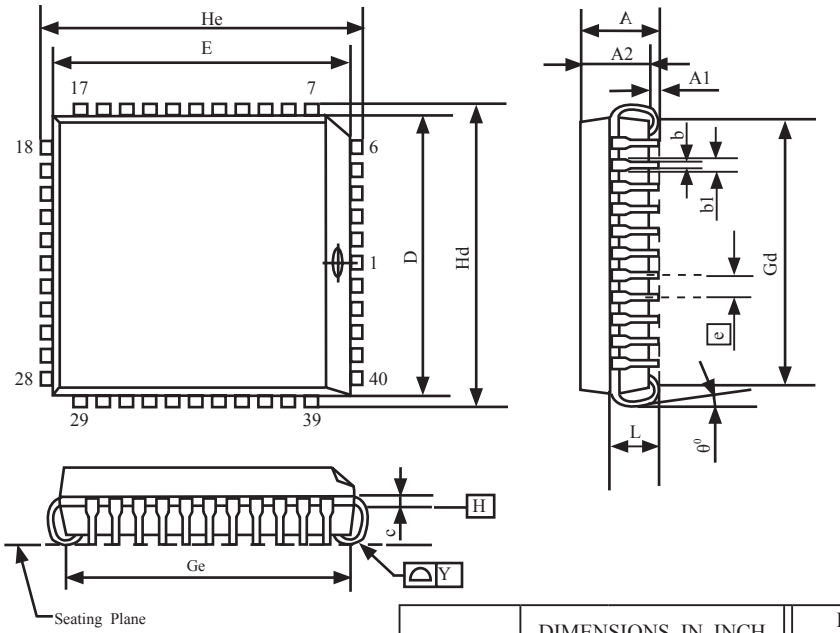
PDIP-40 OUTLINE PACKAGE



SYMBOLS	MIN	NOR	MAX
A	-	-	0.190
A1	0.015	-	-
A2	0.015	0.155	0.160
C	0.008	-	0.015
D	2.025	2.060	2.070
E		0.600	
E1	0.540	0.545	0.550
L	0.120	0.130	0.140
e_b	0.630	0.650	0.670
0	0	7	15

NOTE:
1.JEDEC OUTLINE :MS-011 AC

PLCC-44 OUTLINE PACKAGE



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

NOTE:

1.JEDEC OUTLINE :M0-047 AC

2.DATUM PLANE H IS LACATED AT THE BOTTOM OF THE MOLD PARTING LINE COINCIDENT WITH WHERE THE LEAD EXITS THE BODY.

3.DIMENSIONS E AND D D0 NOT INCLUDE MODE PROTRUSION. ALLOWABLE PROTRUSION IS 10 MIL PRE SIDE.DIMENSIONS E AND D D0 INCLUDE MOLD MISMATCH AND ARE DETERMINED AT DATUM PLANE H.

4.DIMENSION b1 DOES NOT INCLUDE DAMBAR PROTRUSION.

Chapter 2 POWER MANAGENMENT, RESET

2.1 Power Management

There are two power saving modes, which are selectable to drive the STC89xx enter power-saving mode by setting the following SFR PCON.

PCON register (Power Control Register)

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	SMOD	SMOD0	-	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. (defaut)
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.
B5 : Reserved
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.

2.1.1 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, WDT and serial port functions. The CPU status is preserved in its entirety: the 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. ALE and PSEN hold at logic high levels. 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, Timer 2 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 (GFO 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 system clock cycles(24 system clock) to complete the reset.

2.1.2 Power Down (PD) Mode

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 only way to wake-up from power-down mode is hardware reset. The power-down mode can be woken-up by RESET pin, external interrupt /INT0~/INT3, RXD pin, T0 pin, T1 pin . 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 example C program demostrates that power-down mode be woken-up by external interrupt .

```
/*--------------------------------------------------------------------*/
/* --- STC MCU International Limited ------------------------------*/
/* --- STC89xx Series MCU wake up Power-Down mode Demo ------------------------*/
/* --- Mobile: (86)13922805190 ------------------------------------*/
/* --- Fax: 86-755-82944243 ---------------------------------------*/
/* --- Tel: 86-755-82948412 ---------------------------------------*/
/* --- Web: www.STCMCU.com ----------------------------------------*/
/* If you want to use the program or the program referenced in the  */
/* article, please specify in which data and procedures from STC    */
/*--------------------------------------------------------------------*/

#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.

```
;*************************************************************
;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
```

2.2 RESET Control

In STC89xx series, there are 5 sources to generate internal reset. They are RESET pin, On-chip power-on-reset, Watch-Dog-Timer, software reset, and On-chip MAX810 POR timing delay.

2.2.1 Reset pin

The RST pin, which is the input to Schmitt Trigger, is input pin for chip reset. A level change of RESET pin have to keep at least 24 cycles plus 10us in order for CPU internal sampling use. When this signal is brought high for at least two machine cycles plus 10 us, the internal registers are loaded with appropriate values for an orderly system start-up. For normal operation, RST is low.

2.2.2 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 2.0V 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.

PCON register (Power Control Register)

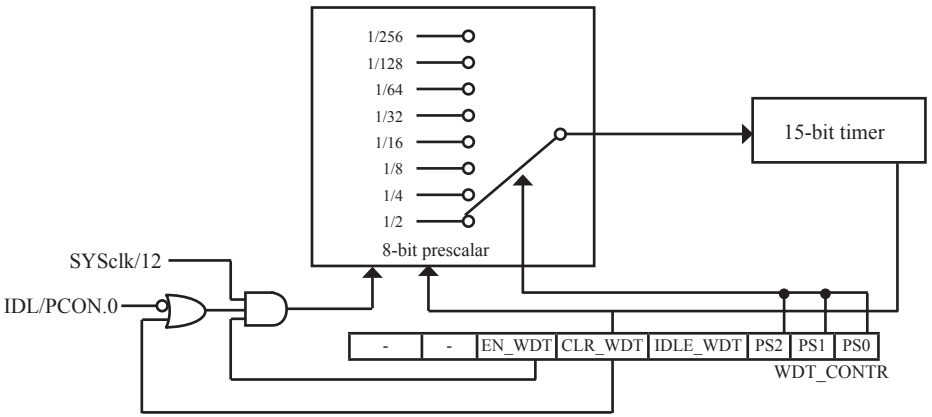
LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	SMOD	SMOD0	-	POF	GF1	GF0	PD	IDL

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

2.2.3 Watch-Dog-Timer

The watch dog timer in STC89xx series MCU 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 STC89xx 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

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	-	-	EN_WDT	CLR_WDT	IDLE_WDT	PS2	PS1	PS0

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 and 12T mode
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 = (N × Pre-scale × 32768) / SYSclk

When MCU in 12T mode, N=12; When MCU in 6T mode, N=6

The SYSclk=20MHz and MCU in 12T mode in the table above.

If SYSclk=12MHz and MCU in 12T mode, 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=12MHz and MCU in 12T mode:

PS2	PS1	PS0	Pre-scale	WDT overflow Time @12MHz and 12T mode
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=11.0592MHz and MCU in 12T mode:

PS2	PS1	PS0	Pre-scale	WDT overflow Time @11.0592MHz and 12T mode
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

The following example is a assembly language program that demostrates STC89xx Series MCU WDT.

```
;/*---------------------------------------------------------------*/
;/* --- STC MCU International Limited ----------------------------*/
;/* --- STC89xx 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 = (N × Pre-scale × 32768) / SYSclk
;When MCU in 12T mode, N=12. When MCU in 6T mode, N=6

WDT_CONTR       EQU     0E1H            ;WDT address
        LED     EQU     P1.5            ;WDT overflow time LED on P1.5
                        ;The WDT overflow time may be measured by the LED light time
Pre_scale_Word  EQU     0x35            ;Start up WDT, Pre-scale value is 64
;SYSclk=18.432, MCU in 12T mode ,WDT overflow time=（12 x 64 x 32768)/18432000 = 1.36 S

                ORG     0000H
                AJMP    MAIN
                ORG     0100H
MAIN:
                CLR     LED                             ;turn LED indicator on
                ACALL   Delay                           ;delay about 1s
                MOV     WDT_CONTR,      #Pre_scale_Word ;Start up WDT
                SETB    LED                             ;turn off LED
WAIT:
                SJMP    WAIT                            ;wait WDT overflow reset
                                                        ;LED will be turned on again after reset

Delay:
                MOV     R0,     #0
                MOV     R1,     #0
                MOV     R2,     #15

Delay_Loop:
                DJNZ    R0,     Delay_Loop
                DJNZ    R1,     Delay_Loop
                DJNZ    R2,     Delay_Loop
                RET
                END
```

2.2.4 Software RESET

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

ISP_CONTR: ISP/IAP Control Register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	ISPEN	SWBS	SWRST	-	-	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.
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.
B4 ~ B3: Reserved.
WT2~WT0 : ISP/IAP waiting time selection while flash is busy.

2.2.5 MAX810 power-on-reset delay

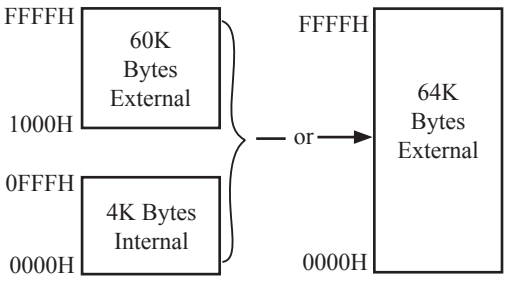
There is another on-chip POR delay circuit is integrated on STC89xx. This circuit is MAX810—sepcial reset circuit and is controlled by configuring flash Option Register. Very long POR delay time – around 400ms will be generated by this circuit once it is enabled.

Chapter 3 Memory Organization

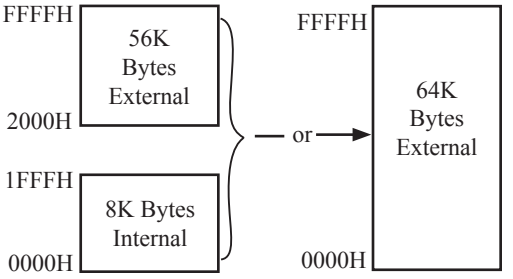
3.1 Program Memory

The STC89xx series MCU has separate address space for Program Memory and Data Memory. Program memory is the memory which stores the program codes for the CPU to execute. There is up to 64K-bytes long of flash memory for program and data storage in the STC89xx series MCU. The lower 4K for the STC89C51RC(8K for the STC89C52RC,16K for STC89C54RC, etc.) may reside on chip. 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 STC89xx series provide 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. STC89xx series 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 STC89xx series follow 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.



STC89C51RC Program Memory



STC89C52RC Program Memory

3.2 Data Memory

3.2.1 On-chip Scratch-Pad RAM

Just the same as the conventional 8051 micro-controller, there are 256 bytes of SRAM data memory including 128 bytes of SFR space available on the STC89xx series. 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 (00H~1FH) 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(refer to section 3.2.4) 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.

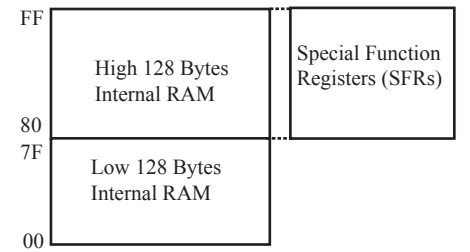
3.2.2 Auxiliary RAM

There are 1024 bytes of additional data RAM available on STC89C54RD+ series (while 256 bytes of additional data RAM are available on STC89C51RC). They may be accessed by the instructions MOVX @Ri or MOVX @DPTR. A control bit – EXTRAM located in AUXR.1 register (refer to section 3.2.4) is to control access of auxiliary RAM. When set, disable the access of auxiliary RAM. When clear (EXTRAM=0), this auxiliary RAM is the default target for the address range from 0x0000 to 0x03FF and can be indirectly accessed by move external instruction, “MOVX @Ri” and “MOVX @DPTR”. If EXTRAM=0 and the target address is over 0x03FF, switches to access external RAM automatically. When EXTRAM=0, the content in DPH is ignored when the instruction MOVX @Ri is executed.

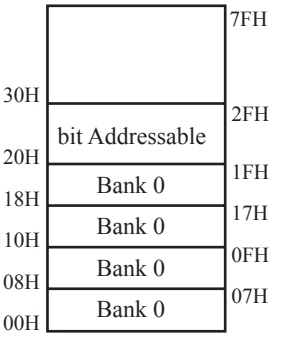
For KEIL-C51 compiler, to assign the variables to be located at Auxiliary RAM, the “pdata” or “xdata” definition should be used. After being compiled, the variables declared by “pdata” and “xdata” will become the memories accessed by “MOVX @Ri” and “MOVX @DPTR”, respectively. Thus the STC89xx series MCU hardware can access them correctly.

3.2.3 External RAM

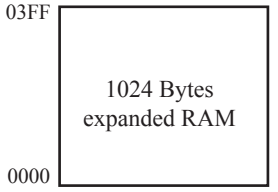
There is 64K-byte addressing space available for STC89xx series to access external data RAM. Just the same as the design in the conventional 8051, the port – P2, P0, ALE, P3.6 and P3.7 have alterative function for external data RAM access. To access the external data memory, the EXTRAM bit should be set to 1. Accesses to external data memory can use either a 16-bit address (using ‘MOVX @DPTR’) or an 8-bit address (using ‘MOVX @Ri’). 8-bit addresses are often used in conjunction with one or more other I/O lines to page the RAM. If an 8-bit address is being used, the contents of the Port 2 SFR remain at the Port 2 pins throughout the external memory cycle. This will facilitate paging access. 16-bit addresses are often used to access up to 64K bytes of external data memory.



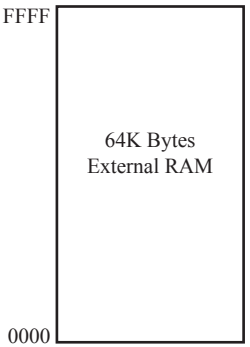
On-chip Scratch-Pad RAM



Lower 128 Bytes of internal SRAM



Auxiliary RAM



External RAM

3.2.4 Special Function Register for RAM

Some SFRs related to RAM are shown as follow.

For fast data movement, STC89xx series support two data pointers. They share the same SFR address and are switched by the register bit – DPS/AUXR1.0.

PSW register

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

CY : Carry flag.
AC : Auxilliary Carry Flag.(For BCD 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.
OV : Overflow flag.
F1 : Flag 1. User-defined flag.
P : Parity flag.

AUXR register

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	-	-	-	-	-	-	EXTRAM	ALEOFF

B7 ~ B3 : resevered.

EXTRAM : Internal/External RAM access

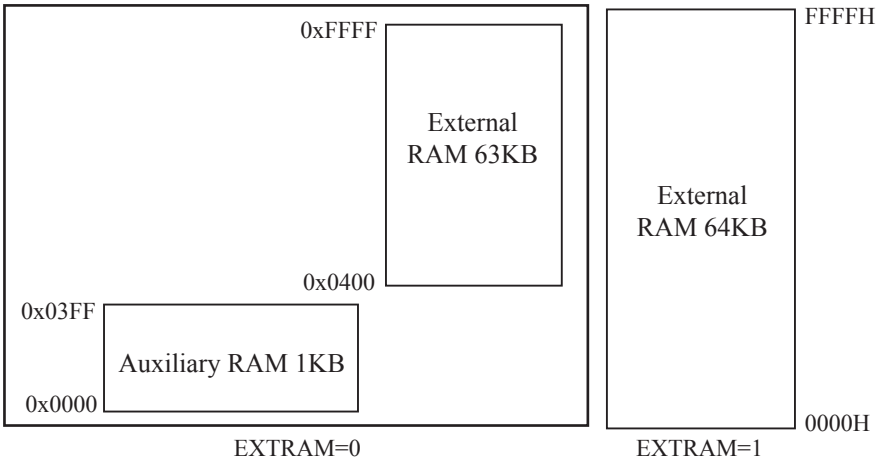
0 : For RD+ series, on-chip auxiliary RAM is enabled and located at the address 0x0000 to 0x03FF. And if address over 0x03FF, off-chip external RAM becomes the target automatically.
For RC seires, on-chip auxiliary RAM is enabled and located at the address 00H to FFH. And if address over FFH, off-chip external RAM becomes the target automatically.

1 : On-chip auxiliary RAM is always disabled.

ALEOFF : Disable/Enable ALE

0 : ALE is emitted at a constant rate of 1/3 the system clock in 6T mode, 1/6 SYSclk in 12T mode.

1 : ALE is active only during a MOVX or MOVC instruction.



AUXR1 register

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	-	-	-	-	GF2	-	-	DPS

GF2 : General purpose user-defined Flag. It can be used by software.

DPS : DPTR registers select bit.

0 : DPTR0 is selected(Default).

1 : The secondary DPTR(DPTR 1) is switched to use.

An example program for internal expanded RAM demo of STC89C58RD+:

```
;/*---------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------*/
;/* --- STC89xx Series MCU internal expanded 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    */
 ;/*---------------------------------------------------------------------*/
#include<reg52.h>
#include<intrins.h>                         /* use _nop_( ) function */

sfr         AUXR    = 0x8e;
sfr         AUXR1   = 0xa2;

sfr         P4          = 0xe8;
sfr         XICON   = 0xc0;

sfr         IPH         = 0xb7;

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

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

void  main ( )
{
            unsigned int array_point = 0;

            /*Test-array: Test_array_one[512], Test_array_two[512] */
            unsigned char xdata Test_array_one[512]  =
            {
                        0x00,     0x01      0x02,     0x03,     0x04      0x05,     0x06,     0x07,
                        0x08,     0x09,     0x0a,     0x0b,     0x0c,     0x0d,     0x0e,     0x0f,
                        0x10,     0x11,     0x12,     0x13,     0x14,     0x15,     0x16,     0x17,
                        0x18,     0x19,     0x1a,     0x1b,     0x1c,     0x1d,     0x1e,     0x1f,
                        0x20,     0x21,     0x22,     0x23,     0x24,     0x25,     0x26,     0x27,
                        0x28,     0x29,     0x2a,     0x2b,     0x2c,     0x2d,     0x2e,     0x2f,
                        0x30,     0x31,     0x32,     0x33,     0x34,     0x35,     0x36,     0x37,
                        0x38,     0x39,     0x3a,     0x3b,     0x3c,     0x3d,     0x3e,     0x3f
```

```
0x40,  0x41,  0x42,  0x43,  0x44,  0x45,  0x46,  0x47,
0x48,  0x49,  0x4a,  0x4b,  0x4c,  0x4d,  0x4e,  0x4f,
0x50,  0x51,  0x52,  0x53,  0x54,  0x55,  0x56,  0x57,
0x58,  0x59,  0x5a,  0x5b,  0x5c,  0x5d,  0x5e,  0x5f,
0x60,  0x61,  0x62,  0x63,  0x64,  0x65,  0x66,  0x67,
0x68,  0x69,  0x6a,  0x6b,  0x6c,  0x6d,  0x6e,  0x6f,
0x70,  0x71,  0x72,  0x73,  0x74,  0x75,  0x76,  0x77,
0x78,  0x79,  0x7a,  0x7b,  0x7c,  0x7d,  0x7e,  0x7f,
0x80,  0x81,  0x82,  0x83,  0x84,  0x85,  0x86,  0x87,
0x88,  0x89,  0x8a,  0x8b,  0x8c,  0x8d,  0x8e,  0x8f,
0x90,  0x91,  0x92,  0x93,  0x94,  0x95,  0x96,  0x97,
0x98,  0x99,  0x9a,  0x9b,  0x9c,  0x9d,  0x9e,  0x9f,
0xa0,  0xa1,  0xa2,  0xa3,  0xa4,  0xa5,  0xa6,  0xa7,
0xa8,  0xa9,  0xaa,  0xab,  0xac,  0xad,  0xae,  0xaf,
0xb0,  0xb1,  0xb2,  0xb3,  0xb4,  0xb5,  0xb6,  0xb7,
0xb8,  0xb9,  0xba,  0xbb,  0xbc,  0xbd,  0xbe,  0xbf,
0xc0,  0xc1,  0xc2,  0xc3,  0xc4,  0xc5,  0xc6,  0xc7,
0xc8,  0xc9,  0xca,  0xcb  ,0xcc,  0xcd,  0xce,  0xcf,
0xd0,  0xd1,  0xd2,  0xd3,  0xd4,  0xd5,  0xd6,  0xd7
0xd8,  0xd9,  0xda,  0xdb,  0xdc,  0xdd,  0xde,  0xdf,
0xe0,  0xe1,  0xe2,  0xe3,  0xe4,  0xe5,  0xe6,  0xe7,
0xe8,  0xe9,  0xea,  0xeb,  0xec,  0xed,  0xee,  0xef,
0xf0,  0xf1,  0xf2,  0xf3,  0xf4,  0xf5,  0xf6,  0xf7,
0xf8,  0xf9,  0xfa,  0xfb,  0xfc,  0xfd,  0xfe,  0xff,
0xff,  0xfe,  0xfd,  0xfc,  0xfb,  0xfa,  0xf9,  0xf8,
0xf7,  0xf6,  0xf5,  0xf4,  0xf3,  0xf2,  0xf1,  0xf0,
0xef,  0xee,  0xed,  0xec,  0xeb,  0xea,  0xe9,  0xe8,
0xe7,  0xe6,  0xe5,  0xe4,  0xe3,  0xe2,  0xe1,  0xe0,
0xdf,  0xde,  0xdd,  0xdc,  0xdb,  0xda,  0xd9,  0xd8,
0xd7,  0xd6,  0xd5,  0xd4,  0xd3,  0xd2,  0xd1,  0xd0,
0xcf,  0xce,  0xcd,  0xcc,  0xcb,  0xca,  0xc9,  0xc8,
0xc7,  0xc6,  0xc5,  0xc4,  0xc3,  0xc2,  0xc1,  0xc0,
0xbf,  0xbe,  0xbd,  0xbc,  0xbb,  0xba,  0xb9,  0xb8,
0xb7,  0xb6,  0xb5,  0xb4,  0xb3,  0xb2,  0xb1,  0xb0,
0xaf,  0xae,  0xad,  0xac,  0xab,  0xaa,  0xa9,  0xa8,
0xa7,  0xa6,  0xa5,  0xa4,  0xa3,  0xa2,  0xa1,  0xa0,
0x9f,  0x9e,  0x9d,  0x9c,  0x9b,  0x9a,  0x99,  0x98,
0x97,  0x96,  0x95,  0x94,  0x93,  0x92,  0x91,  0x90,
0x8f,  0x8e,  0x8d,  0x8c,  0x8b,  0x8a,  0x89,  0x88,
0x87,  0x86,  0x85,  0x84,  0x83,  0x82,  0x81,  0x80,
0x7f,  0x7e,  0x7d,  0x7c,  0x7b,  0x7a,  0x79,  0x78,
0x77,  0x76,  0x75,  0x74,  0x73,  0x72,  0x71,  0x70,
0x6f,  0x6e,  0x6d,  0x6c,  0x6b,  0x6a,  0x69,  0x68,
0x67,  0x66,  0x65,  0x64,  0x63,  0x62,  0x61,  0x60,
0x5f,  0x5e,  0x5d,  0x5c,  0x5b,  0x5a,  0x59,  0x58,
0x57,  0x56,  0x55,  0x54,  0x53,  0x52,  0x51,  0x50,
0x4f,  0x4e,  0x4d,  0x4c,  0x4b,  0x4a,  0x49,  0x48,
0x47,  0x46,  0x45,  0x44,  0x43,  0x42,  0x41,  0x40,
```

```
        0x3f,   0x3e,   0x3d,   0x3c,   0x3b,   0x3a,   0x39,   0x38,
        0x37,   0x36,   0x35,   0x34,   0x33,   0x32,   0x31,   0x30,
        0x2f,   0x2e,   0x2d,   0x2c,   0x2b,   0x2a,   0x29,   0x28,
        0x27,   0x26,   0x25,   0x24,   0x23,   0x22,   0x21,   0x20,
        0x1f,   0x1e,   0x1d,   0x1c,   0x1b,   0x1a,   0x19,   0x18,
        0x17,   0x16,   0x15,   0x14,   0x13,   0x12,   0x11,   0x10,
        0x0f,   0x0e,   0x0d,   0x0c,   0x0b,   0x0a,   0x09,   0x08,
        0x07,   0x06,   0x05,   0x04,   0x03,   0x02,   0x01,   0x00
};

unsigned char xdata Test_array_two[512] =
{
        0x00,   0x01    0x02,   0x03,   0x04    0x05,   0x06,   0x07,
        0x08,   0x09,   0x0a,   0x0b,   0x0c,   0x0d,   0x0e,   0x0f,
        0x10,   0x11,   0x12,   0x13,   0x14,   0x15,   0x16,   0x17,
        0x18,   0x19,   0x1a,   0x1b,   0x1c,   0x1d,   0x1e,   0x1f,
        0x20,   0x21,   0x22,   0x23,   0x24,   0x25,   0x26,   0x27,
        0x28,   0x29,   0x2a,   0x2b,   0x2c,   0x2d,   0x2e,   0x2f,
        0x30,   0x31,   0x32,   0x33,   0x34,   0x35,   0x36,   0x37,
        0x38,   0x39,   0x3a,   0x3b,   0x3c,   0x3d,   0x3e,   0x3f
        0x40,   0x41,   0x42,   0x43,   0x44,   0x45,   0x46,   0x47,
        0x48,   0x49,   0x4a,   0x4b,   0x4c,   0x4d,   0x4e,   0x4f,
        0x50,   0x51,   0x52,   0x53,   0x54,   0x55,   0x56,   0x57,
        0x58,   0x59,   0x5a,   0x5b,   0x5c,   0x5d,   0x5e,   0x5f,
        0x60,   0x61,   0x62,   0x63,   0x64,   0x65,   0x66,   0x67,
        0x68,   0x69,   0x6a,   0x6b,   0x6c,   0x6d,   0x6e,   0x6f,
        0x70,   0x71,   0x72,   0x73,   0x74,   0x75,   0x76,   0x77,
        0x78,   0x79,   0x7a,   0x7b,   0x7c,   0x7d,   0x7e,   0x7f,
        0x80,   0x81,   0x82,   0x83,   0x84,   0x85,   0x86,   0x87,
        0x88,   0x89,   0x8a,   0x8b,   0x8c,   0x8d,   0x8e,   0x8f,
        0x90,   0x91,   0x92,   0x93,   0x94,   0x95,   0x96,   0x97,
        0x98,   0x99,   0x9a,   0x9b,   0x9c,   0x9d,   0x9e,   0x9f,
        0xa0,   0xa1,   0xa2,   0xa3,   0xa4,   0xa5,   0xa6,   0xa7,
        0xa8,   0xa9,   0xaa,   0xab,   0xac,   0xad,   0xae,   0xaf,
        0xb0,   0xb1,   0xb2,   0xb3,   0xb4,   0xb5,   0xb6,   0xb7,
        0xb8,   0xb9,   0xba,   0xbb,   0xbc,   0xbd,   0xbe,   0xbf,
        0xc0,   0xc1,   0xc2,   0xc3,   0xc4,   0xc5,   0xc6,   0xc7,
        0xc8,   0xc9,   0xca,   0xcb    ,0xcc,  0xcd,   0xce,   0xcf,
        0xd0,   0xd1,   0xd2,   0xd3,   0xd4,   0xd5,   0xd6,   0xd7
        0xd8,   0xd9,   0xda,   0xdb,   0xdc,   0xdd,   0xde,   0xdf,
        0xe0,   0xe1,   0xe2,   0xe3,   0xe4,   0xe5,   0xe6,   0xe7,
        0xe8,   0xe9,   0xea,   0xeb,   0xec,   0xed,   0xee,   0xef,
        0xf0,   0xf1,   0xf2,   0xf3,   0xf4,   0xf5,   0xf6,   0xf7,
        0xf8,   0xf9,   0xfa,   0xfb,   0xfc,   0xfd,   0xfe,   0xff,
        0xff,   0xfe,   0xfd,   0xfc,   0xfb,   0xfa,   0xf9,   0xf8,
        0xf7,   0xf6,   0xf5,   0xf4,   0xf3,   0xf2,   0xf1,   0xf0,
        0xef,   0xee,   0xed,   0xec,   0xeb,   0xea,   0xe9,   0xe8,
        0xe7,   0xe6,   0xe5,   0xe4,   0xe3,   0xe2,   0xe1,   0xe0,
```

```
                0xdf,   0xde,   0xdd,   0xdc,   0xdb,   0xda,   0xd9,   0xd8,
                0xd7,   0xd6,   0xd5,   0xd4,   0xd3,   0xd2,   0xd1,   0xd0,
                0xcf,   0xce,   0xcd,   0xcc,   0xcb,   0xca,   0xc9,   0xc8,
                0xc7,   0xc6,   0xc5,   0xc4,   0xc3,   0xc2,   0xc1,   0xc0,
                0xbf,   0xbe,   0xbd,   0xbc,   0xbb,   0xba,   0xb9,   0xb8,
                0xb7,   0xb6,   0xb5,   0xb4,   0xb3,   0xb2,   0xb1,   0xb0,
                0xaf,   0xae,   0xad,   0xac,   0xab,   0xaa,   0xa9,   0xa8,
                0xa7,   0xa6,   0xa5,   0xa4,   0xa3,   0xa2,   0xa1,   0xa0,
                0x9f,   0x9e,   0x9d,   0x9c,   0x9b,   0x9a,   0x99,   0x98,
                0x97,   0x96,   0x95,   0x94,   0x93,   0x92,   0x91,   0x90,
                0x8f,   0x8e,   0x8d,   0x8c,   0x8b,   0x8a,   0x89,   0x88,
                0x87,   0x86,   0x85,   0x84,   0x83,   0x82,   0x81,   0x80,
                0x7f,   0x7e,   0x7d,   0x7c,   0x7b,   0x7a,   0x79,   0x78,
                0x77,   0x76,   0x75,   0x74,   0x73,   0x72,   0x71,   0x70,
                0x6f,   0x6e,   0x6d,   0x6c,   0x6b,   0x6a,   0x69,   0x68,
                0x67,   0x66,   0x65,   0x64,   0x63,   0x62,   0x61,   0x60,
                0x5f,   0x5e,   0x5d,   0x5c,   0x5b,   0x5a,   0x59,   0x58,
                0x57,   0x56,   0x55,   0x54,   0x53,   0x52,   0x51,   0x50,
                0x4f,   0x4e,   0x4d,   0x4c,   0x4b,   0x4a,   0x49,   0x48,
                0x47,   0x46,   0x45,   0x44,   0x43,   0x42,   0x41,   0x40,
                0x3f,   0x3e,   0x3d,   0x3c,   0x3b,   0x3a,   0x39,   0x38,
                0x37,   0x36,   0x35,   0x34,   0x33,   0x32,   0x31,   0x30,
                0x2f,   0x2e,   0x2d,   0x2c,   0x2b,   0x2a,   0x29,   0x28,
                0x27,   0x26,   0x25,   0x24,   0x23,   0x22,   0x21,   0x20,
                0x1f,   0x1e,   0x1d,   0x1c,   0x1b,   0x1a,   0x19,   0x18,
                0x17,   0x16,   0x15,   0x14,   0x13,   0x12,   0x11,   0x10,
                0x0f,   0x0e,   0x0d,   0x0c,   0x0b,   0x0a,   0x09,   0x08,
                0x07,   0x06,   0x05,   0x04,   0x03,   0x02,   0x01,   0x00
        };
        ERROR_LED = 1;
        OK_LED = 1;
        for (array_point = 0; array_point<512; array_point++)
        {
                 if (Test_array_one[array_point] != Test_array_two [array_point])
                {
                        ERROR_LED = 0;
                        OK_LED = 1;
                        break;
                }
                else{
                        OK_LED = 0;
                        ERROR_LED = 1;
                    }
        }
        while (1);
}
```

Chapter 4 Configurable I/O Ports Configurations

The STC89xx series MCU has following I/O ports: P0.0~P0.7, P1.0~P1.7, P2.0~P2.7, P3.0~P3.7, P4.0~P4.3(only be available to LQFP44 and PLCC44). All port pins on STC89xx series may be independently configured to one of four modes : 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.

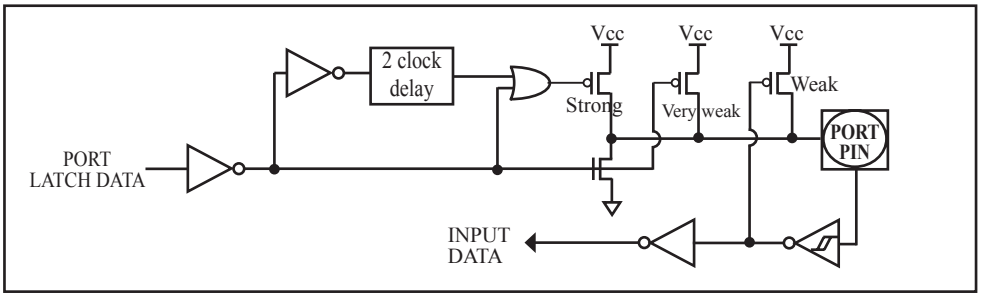
4.1 Quasi-bidirectional I/O

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.

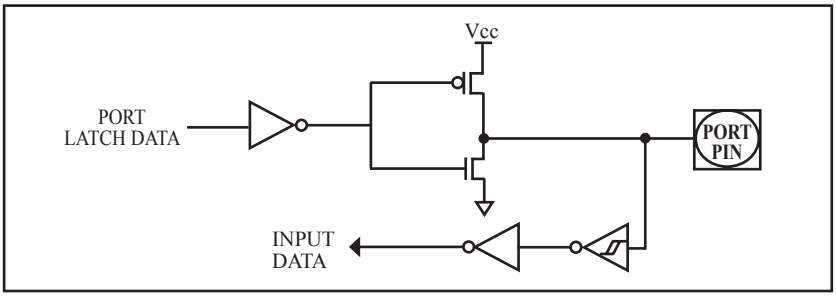
The third pull-up is referred to as the “strong” pull-up. This pull-up is used to speed up low-to-high transitions on a quasi-bidirectional port pin when the port register changes from a logic “0” to a logic “1”. When this occurs, the strong pull-up turns on for two CPU clocks, quickly pulling the port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



Quasi-bidirectional output

4.2 Push-pull Output

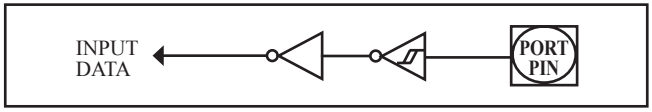
The push-pull output configuration has the same pull-down structure as both the open-drain and the quasi-bidirectional output modes, but provides a continuous strong pull-up when the port register conatins a logic “1”. The push-pull mode may be used when more source current is needed from a port output.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



Push-pull output

4.3 Input-only Mode

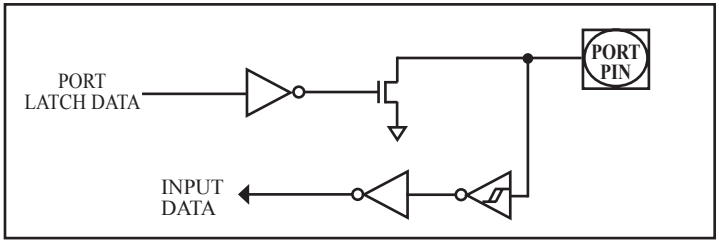
The input-only configuration is a Schmitt-triggered input without any pull-up resistors on the pin.



Input-only Mode

4.4 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 pull-down for this mode is the same as for the quasi-bidirectional mode. The input path of the port pin in this configuration is the same as quasi-bidirection mode.



Open-drain output

Chapter 5 Instruction System

5.1 Special Function Registers

	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	
0F8H									0FFH
0F0H	B 0000,0000								0F7H
0E8H	P4 xxxx,1111								0EFH
0E0H	ACC 0000,0000	WDT_CONR xx00,0000	ISP_DATA 1111,1111	ISP_ADDRH 0000,0000	ISP_ADDRL 0000,0000	ISP_CMD xxxx,x000	ISP_TRIG xxxx,xxxx	ISP_CONTR 000x,x000	0E7H
0D8H									0DFH
0D0H	PSW 0000,0000								0D7H
0C8H	T2CON 0000,0000	T2MOD xxxx,xx00	RCAP2L 0000,0000	RCAP2H 0000,0000	TL2 0000,0000	TH2 0000,0000			0CFH
0C0H	XICON 0000,0000								0C7H
0B8H	IP x0x0,0000	SADEN 0000,0000							0BFH
0B0H	P3 1111,1111							IPH 0000,0000	0B7H
0A8H	IE 0000,0000	SADDR 0000,0000							0AFH
0A0H	P2 1111,1111		AUXR1 xxxx,0xx0					Don't use	0A7H
098H	SCON 0000,0000	SBUF xxxx,xxxx							09FH
090H	P1 1111,1111								097H
088H	TCON 0000,0000	TMOD 0000,0000	TL0 0000,0000	TL1 0000,0000	TH0 0000,0000	TH1 0000,0000	AUXR xxxx,xx00		08FH
080H	P0 1111,1111	SP 0000,0111	DPL 0000,0000	DPH 0000,0000				PCON 00x1,0000	087H
	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	

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
PCON	Power Control	87H	SMOD	SMOD0	-	POF	GF1	GF0	PD	IDL	00x1 0000B
TCON	Timer/Counter 0 and 1 Control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	0000 0000B
TMOD	Timer/Counter 0 and 1 Modes	89H	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	0000 0000B
TL0	Timer/Counter 0 Low Byte	8AH									0000 0000B
TL1	Timer/Counter 1 Low Byte	8BH									0000 0000B
TH0	Timer/Counter 0 High Byte	8CH									0000 0000B
TH1	Timer/Counter 1 High Byte	8DH									0000 0000B
AUXR	Auxiliary register 0	8EH	-	-	-	-	-	-	EXTRAM	ALEOFF	xxxx xx00B
P1	Port 1	90H	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	1111 1111B
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
AUXR1	Auxiliary register1	A2H	-	-	-	-	GF2		-	-	DPS	0xxx 0xx0B
IE	Interrupt Enable	A8H	EA	-	ET2	ES	ET1	EX1	ET0	EX0	0000 0000B
SADDR	Slave Address	A9H									0000 0000B
P3	Port 3	B0H	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0	1111 1111B
IPH	Interrupt Priority High	B7H	PX3H	PX2H	PT2H	PSH	PT1H	PX1H	PT0H	PX0H	0000 0000B
IP	Interrupt Priority Low	B8H	-	-	PT2	PS	PT1	PX1	PT0	PX0	xx00 0000B
SADEN	Slave Address Mask	B9H									0000 0000B
XICON	Auxiliary Interrupt Control	C0H	PX3	EX3	IE3	IT3	PX2	EX2	IE2	IT2	0000 0000B
T2CON	Timer/Counter 2 Control	C8H	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$	0000 0000B
T2MOD	Timer/Counter 2 Mode	C9H	-	-	-	-	-	-	T2OE	DCEN	xxxx xx00B
RCAP2L	Timer/Counter 2 Reload/Capture Low Byte	CAH									0000 0000B

Symbol	Description	Address	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
RCAP2H	Timer/Counter 2 Reload/Capture High Byte	CBH									0000 0000B
TL2	Timer/Counter Low Byte	CCH									0000 0000B
TH2	Timer/Counter High Byte	CDH									0000 0000B
PSW	Program Status Word	D0H	CY	AC	F0	RS1	RS0	OV	F1	P	0000 0000B
ACC	Accumulator	E0H									0000 0000B-
WDT_CONTR	Watch-Dog-Timer Control Register	E1H	-	-	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	-	-	-	-	-	MS2	MS1	MS0	xxxx x000B
ISP_TRIG	ISP/IAP Flash Command Trigger	E6H									xxxx xxxxB
ISP_CONTR	ISP/IAP Control Register	E7H	ISPEN	SWBS	SWRST	-	-	WT2	WT1	WT0	000x x000B
P4	Port 4	E8H	P4.7	P4.6	P4.5	P4.4	P4.3	P4.2	P4.1	P4.0	1111 1111B
B	B Register	F0H									0000 0000B

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. This causes the stack to begin at location 08H.

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

bit	7	6	5	4	3	2	1	0
name	CY	AC	F0	RS1	RS0	OV	F1	P

CY : Carry flag.
AC : Auxilliary Carry Flag.(For BCD 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.
OV : Overflow flag.
F1 : Flag 1. User-defined flag.
P : Parity flag.

Data Pointer

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.

For fast data movement, STC89xx series support two data pointers. They share the same SFR address and are switched by the register bit – DPS/AUXR1.0.

AUXR1 register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	-	-	-	-	GF2	-	-	DPS

GF2 : General purpose user defined flag. It can be used by software.

DPS : DPTR registers select bit.
0 : DPTR0 is selected(Default).
1 : The secondary DPTR(DPTR 1) is switched to use.

The following program is an assembly program that demostrates how the dual data pointer be used.

```
AUXR1       DATA    0A2H        ;Define special function register AUXR1
MOV         AUXR1,  #0          ;DPS=0, select DPTR0

MOV         DPTR,   #1FFH       ;Set DPTR0 for 1FFH
MOV         A,      #55H
MOVX        @DPTR,  A           ;load the value 55H in the 1FFH unit

MOV         DPTR,   #2FFH       ;Set DPTR0 for 2FFH
MOV         A,      #0AAH
MOVX        @DPTR,  A           ;load the value 0AAH in the 2FFH unit

INC         AUXR1               ;DPS=1, DPTR1 is selected
MOV         DPTR,   #1FFH       ;Set DPTR1 for 1FFH
MOVX        A,      @DPTR       ;Get the content of 1FFH unit
                                ;which is pointed by DPTR1,
                                ;the content of Accumulator has changed for 55H
INC         AUXR1               ;DPS=0, DPTR0 is selected
MOVX        A,      @DPTR       ;Get the content of 2FFH unit
                                ;which is pointed by DPTR0,
                                ;the content of Accumulator has changed for 0AAH
INC         AUXR1               ;DPS=1, DPTR1 is selected
MOVX        A,      @DPTR       ;Get the content of 1FFH unit
                                ;which is pointed by DPTR1,
                                ;the content of Accumulator has changed for 55H
INC         AUXR1               ;DPS=0, DPTR0 is selected
MOVX        A,      @DPTR       ;Get the content of 2FFH unit
                                ;which is pointed by DPTR0,
                                ;the content of Accumulator has changed for 0AAH
```

5.2 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 eight modes available:

- Register
- Direct
- Indirect
- Immediate
- Relative
- Absolute
- Long
- Indexed

Direct Addressing(DIR)
In direct addressing the operand is specified by an 8-bit address field in the instruction. Only 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.

Immediate Constant(IMM)
The value of a constant can follow the opcode in the program memory.

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.

5.3 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 instructions. Once you are familiar with the instruction set, this chart should prove a handy and quick source of reference.

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

All arithmetic instructions execute one machine cycle except the INC DPTR instruction (two machine cycles) and the MUL AB and DIV AB instructions (four machine cycles). Note that a machine cycle contains 12 clocks and takes 1us if MCU is operating in 12T mode and from a 12MHz clock.While MCU is operating in 6T mode, a machine cycle contains 6 clocks.

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

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

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

5.4 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.

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

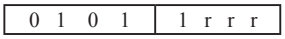
will clear bits 7, 3, and 2 of output port 1.

ANL A,Rn

Bytes: 1

Cycles: 1

Encoding:



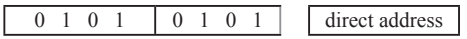
Operation: ANL
(A)←**(A)** ∧ (Rn)

ANL A,direct

Bytes: 2

Cycles: 1

Encoding:



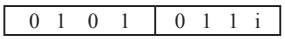
Operation: ANL
(A)←**(A)** ∧ (direct)

ANL A,@Ri

Bytes: 1

Cycles: 1

Encoding:



Operation: ANL
(A)←**(A)** ∧ ((Ri))

ANL A,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 0 1	0 1 0 0	immediate data

Operation: ANL
(A)←(A) ∧ #data

ANL direct,A

Bytes: 2

Cycles: 1

Encoding:

0 1 0 1	0 0 1 0	direct address

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

ANL direct,#data

Bytes: 3

Cycles: 2

Encoding:

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

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

ANL C , <src-bit>

Function: Logical-AND for bit variables

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

Only direct addressing is allowed for the source operand.

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

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

ANL C,bit

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	0 0 1 0	bit address

Operation: ANL
(C) ← (C) ∧ (bit)

ANL C, /bit

Bytes: 2

Cycles: 2

Encoding:

1 0 1 1	0 0 0 0	bit address

Operation: ADD

(C)←(C) ∧ $\overline{\text{(bit)}}$

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

Function: Compare and Jump if Not Equal

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

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

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

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

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

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

WAIT: CJNE A,P1,WAIT

clears the carry flag and continues with the next instruction in sequence, since the Accumulator does equal the data read from P1. (If some other value was being input on 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:

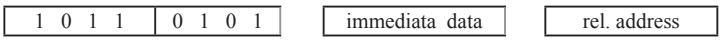
(PC) ← (PC) + 3
IF (A) < > *(direct)*
THEN
    (PC) ← (PC) + *relative offset*
IF (A) < *(direct)*
THEN
    (C) ← 1
ELSE
    (C) ← 0

CJNE A,#data,rel

Bytes: 3

Cycles: 2

Encoding:



Operation:

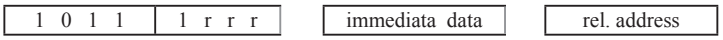
(PC) ← **(PC) + 3**
IF (A) < > *(data)*
THEN
(PC) ← **(PC) +** *relative offset*
IF (A) < *(data)*
THEN
(C) ← **1**
ELSE
(C) ← **0**

CJNE Rn,#data,rel

Bytes: 3

Cycles: 2

Encoding:



Operation:

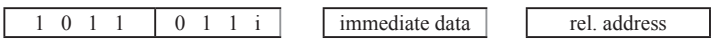
(PC) ← **(PC) + 3**
IF (Rn) < > *(data)*
THEN
(PC) ← **(PC) +** *relative offset*
IF (Rn) < *(data)*
THEN
(C) ← **1**
ELSE
(C) ← **0**

CJNE @Ri,#data,rel

Bytes: 3

Cycles: 2

Encoding:



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,

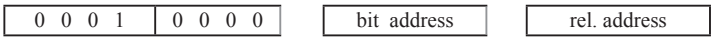
```
JBC     ACC.3, LABEL1
JBC     ACC.2, LABEL2
```

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,

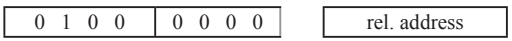
```
JC      LABEL1
CPL     C
JC      LABEL2s
```

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

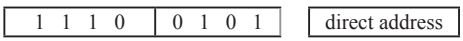
Operation: MOV
(A) ← (Rn)

***MOV A,direct**

Bytes: 2

Cycles: 1

Encoding:



Operation: MOV
(A)← (direct)

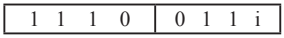
***MOV A, ACC is not a valid instruction**

MOV A,@Ri

Bytes: 1

Cycles: 1

Encoding:



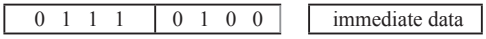
Operation: MOV
(A) ← ((Ri))

MOV A,#data

Bytes: 2

Cycles: 1

Encoding:



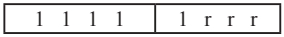
Operation: MOV
(A)← #data

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:

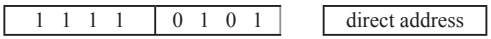
Operation: MOV
(Rn) ← #data

MOV direct, A

Bytes: 2

Cycles: 1

Encoding:



Operation: MOV
(direct) ← (A)

MOV direct, Rn

Bytes: 2

Cycles: 2

Encoding:

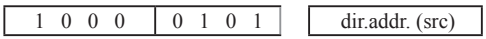
Operation: MOV
(direct) ← (Rn)

MOV direct, direct

Bytes: 3

Cycles: 2

Encoding:



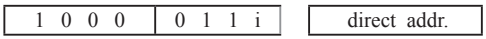
Operation: MOV
(direct)← (direct)

MOV direct, @Ri

Bytes: 2

Cycles: 2

Encoding:



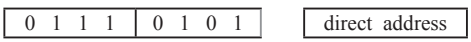
Operation: MOV
(direct)←((Ri))

MOV direct,#data

Bytes: 3

Cycles: 2

Encoding:



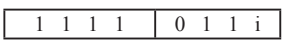
Operation: MOV
(direct) ← #data

MOV @Ri, A

Bytes: 1

Cycles: 1

Encoding:



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)**∨$\overline{\text{(bit)}}$

POP direct

Function: Pop from stack

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

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

```
POP     DPH
POP     DPL
```

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

```
POP     SP
```

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

Bytes: 2

Cycles: 2

Encoding:

1 1 0 1	0 0 0 0	direct address

Operation: POP

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

RLC A

Function: Rotate Accumulator Left through the Carry flag

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

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

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	0	0	1	1

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

RR A

Function: Rotate Accumulator Right

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

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

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	1	1

Operation:
RR
(An) ← (An+1) n = 0 - 6
(A7) ← (A0)

RRC A

Function: Rotate Accumulator Right through the Carry flag

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

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

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	0	1	1

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

SETB <bit>

Function: Set bit

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

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

SETB C

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	0	1	1

Operation: SETB
(C) ← 1

SETB bit

Bytes: 2

Cycles: 1

Encoding:

1	1	0	1	0	0	1	0	bit address

Operation: SETB
(bit) ← 1

SJMP rel

Function: Short Jump

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

Example: The label "RELADR" is assigned to an instruction at program memory location 0123H. The instruction,
SJMP RELADR
will assemble into location 0100H. After the instruction is executed, the PC will contain the value 0123H.
(*Note:* Under the above conditions the instruction following SJMP will be at 102H.Therefore, the displacement byte of the instruction will be the relative offset (0123H - 0102H) = 21H. Put another way, an SJMP with a displacement of 0FEH would be an one-instruction infinite loop).

Bytes: 2

Cycles: 2

Encoding:

1	0	0	0	0	0	0	0	rel. address

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

SUBB A, <src-byte>

Function: Subtract with borrow

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

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

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

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

SUBB A, R2

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

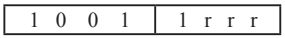
Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the carry (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.

SUBB A, Rn

Bytes: 1

Cycles: 1

Encoding:



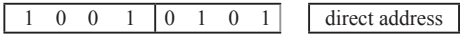
Operation: SUBB
(A) ← (A) - (C) - (Rn)

SUBB A, direct

Bytes: 2

Cycles: 1

Encoding:



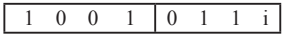
Operation: SUBB
(A) ← (A) - (C) - (direct)

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) $\rightleftarrows$ (Rn)

XCH A, direct

Bytes: 2

Cycles: 1

Encoding:

1 1 0 0	0 1 0 1	direct address

Operation: XCH

(A) $\rightleftarrows$ (direct)

XCH A, @Ri

Bytes: 1

Cycles: 1

Encoding:

1 1 0 0	0 1 1 i

Operation: XCH

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

XCHD A, @Ri

Function: Exchange Digit

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

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

XCHD A, @R0

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

Bytes: 1

Cycles: 1

Encoding:

1 1 0 1	0 1 1 i

Operation: XCHD

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

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

Function: Logical Exclusive-OR for byte variables

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

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

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

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

XRL A, R0

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

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

XRL P1, #00110001B

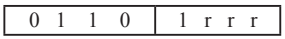
will complement bits 5,4 and 0 of outpue Port 1.

XRL A, Rn

Bytes: 1

Cycles: 1

Encoding:



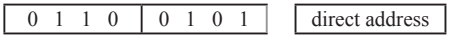
Operation: XRL
(A) ← (A) ⅄ (Rn)

XRL A, direct

Bytes: 2

Cycles: 1

Encoding:



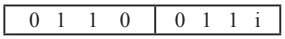
Operation: XRL
(A) ← (A) ⅄ (direct)

XRL A, @Ri

Bytes: 1

Cycles: 1

Encoding:



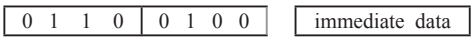
Operation: XRL
(A) ← (A) ⅄ ((Ri))

XRL A, #data

Bytes: 2

Cycles: 1

Encoding:



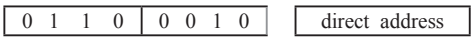
Operation: XRL
(A) ← (A) ⅄ #data

XRL direct, A

Bytes: 2

Cycles: 1

Encoding:



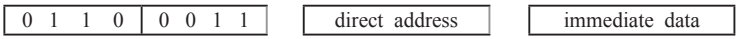
Operation: XRL
(direct) ← (direct) ⅄ (A)

XRL direct, #dataw

Bytes: 3

Cycles: 2

Encoding:



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

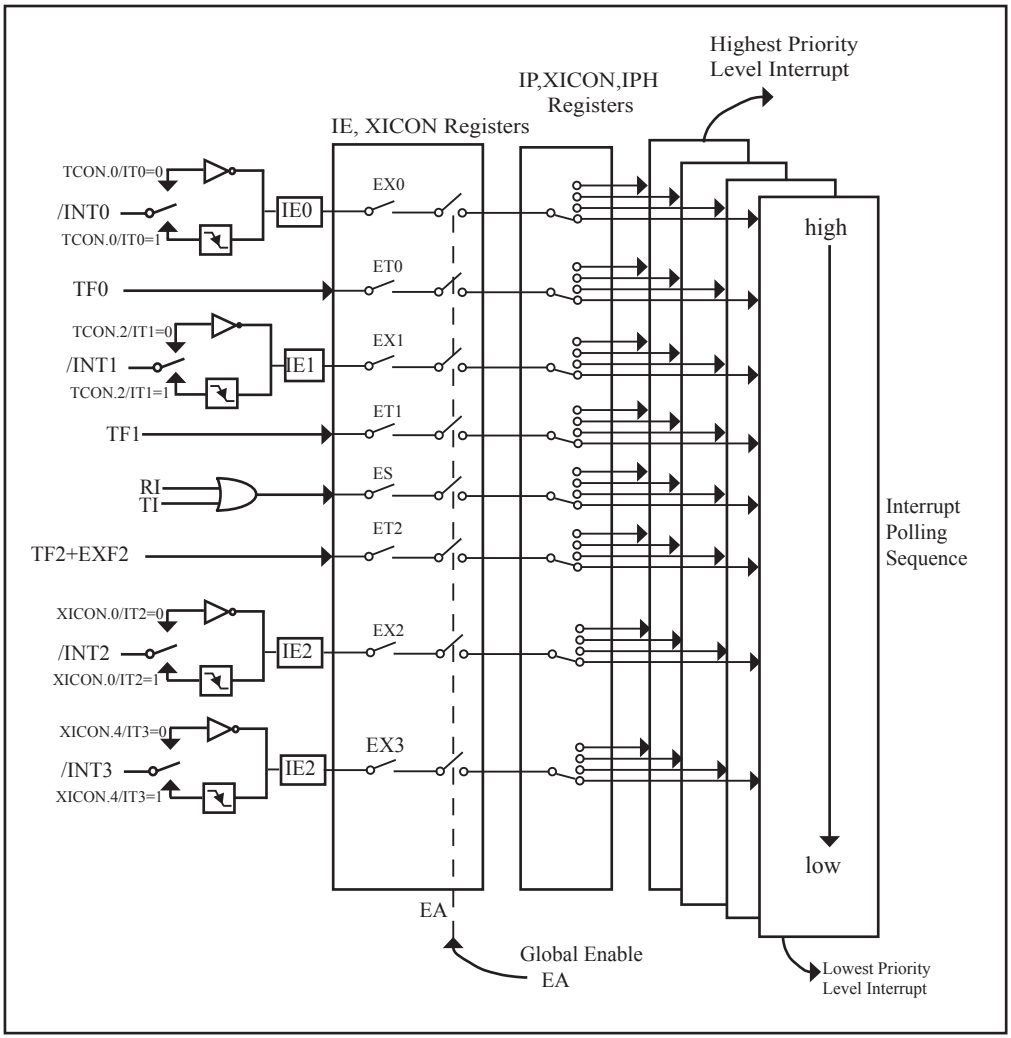
Chapter 6 Interrupt

There are 8 interrupt vector addresses available in STC89xx series. Associating with each interrupt vector, each interrupt source can be individually enabled or disabled by setting or clearing a bit in the registers IE and XICON. The register also contains a global disable bit(EA), which can be cleared to disable all interrupts at once.

Each interrupt source has two corresponding bits to represent its priority. One is located in SFR named IPH and the other in IP or XICON 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 Source	Vector address	Polling Sequence	Interrupt Priority setting (IP/XICON,IPH)	Priority 0 (lowest)	Priority 1	Priority 2	Priority 3 (highest)	Interrupt Request	Interrupt Enable Control Bit
/INT0 (External interrupt 0)	0003H	0(highest)	PX0H,PX0	0,0	0,1	1,0	1,1	IE0	EX0/EA
Timer 0	000BH	1	PT0H,PT0	0,0	0,1	1,0	1,1	TF0	ET0/EA
/INT1 (External interrupt 1)	0013H	2	PX1H,PX1	0,0	0,1	1,0	1,1	IE1	EX1/EA
Timer1	001BH	3	PT1H,PT1	0,0	0,1	1,0	1,1	TF1	ET1/EA
UART (Serial Interface)	0023H	4	PSH,PS	0,0	0,1	1,0	1,1	RI+TI	ES/EA
Timer 2	002BH	5	PT2H,PT2	0,0	0,1	1,0	1,1	TF2+EXF2	ET2/EA
/INT2	0033H	6	PX2H,PX2	0,0	0,1	1,0	1,1	IE2	EX2/EA
/INT3	003BH	7(lowest)	PX3H,PX3	0,0	0,1	1,0	1,1	IE3	EX3/EA

6.1 Interrupt Structure



Interrupt system diagram of STC89xx series

The External Interrupts $\overline{INT0}$, $\overline{INT1}$, $\overline{INT2}$ and $\overline{INT3}$ can each be either level-activated or transition-activated, depending on bits IT0 and IT1 in Register TCON, IT2 and IT3 in SFR XICON. The flags that actually generate these interrupts are bits IE0 and IE1 in TCON, IE2 and IE3 in XICON. 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 Timer 2 Interrupt is generates by the logical OR of TF2 and EXF2. Just the same as serial port, neither of these flags is cleared by hardware when the service routine is vectored to.

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	-	ET2	ES	ET1	EX1	ET0	EX0	0000 0000B
IP	Interrupt Priority Low	B8H	-	-	PT2	PS	PT1	PX1	PT0	PX0	xx00 0000B
IPH	Interrupt Priority High	B7H	PX3H	PX2H	PT2H	PSH	PT1H	PX1H	PT0H	PX0H	0000,0000B
TCON	Timer/Counter 0 and 1 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
T2CON	Timer/Counter 2 Control	C8H	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$	0000 0000B
XICON	Auxiliary Interupt Control	C0H	PX3	EX3	IE3	IT3	PX2	EX2	IE2	IT1	0000 0000B

IE: Interrupt Enable Rsgister

(MSB) (LSB)

EA	-	ET2	ES	ET1	EX1	ET0	EX0

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

Symbol	Position	Function
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.
ET2	IE.5	Timer 2 interrupt enable bit
ES	IE.4	Serial Port interrupt enable bit
ET1	IE.3	Timer 1 interrupt enable bit
EX1	IE.2	External interrupt 1 enable bit
ET0	IE.1	Timer 0 interrupt enable bit
EX0	IE.0	External interrupt 0 enable bit

IP: Interrupt Priority Low Register

(MSB) (LSB)

-	-	PT2	PS	PT1	PX1	PT0	PX0

Priority bit = 1 assigns high priority .
Priority bit = 0 assigns low priority.

Symbol	Position	Function
PT2	IP.5	Timer 2 interrupt priority bit
PS	IP.4	Serial Port interrupt priority bit.
PT1	IP.3	Timer 1 interrupt priority bit
PX1	IP.2	External interrupt 1 priority bit
PT0	IP.1	Timer 0 interrupt priority bit
PX0	IP.0	External interrupt 0 priority bit

IPH: Interrupt Priority High Register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	PX3H	PX2H	PT2H	PSH	PT1H	PX1H	PT0H	PX0H

PX3H : If set, Set priority for external interrupt 3 highest
PX2H : If set, Set priority for external interrupt 2 highest
PT2H : If set, Set for Timer 2 interrupt highest
PSH : If set, Set priority for serial port highest
PT1H : If set, Set priority for Timer 1 interrupt highest
PX1H : If set, Set priority for external interrupt 1 highest
PT0H : If set, Set priority for Timer 0 interrupt highest
PX0H : If set, Set priority for external interrupt 0 highest

TCON register: Timer/Counter Control Register

(MSB) (LSB)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Symbol	Position	Name and Significance
TF1	TCON.7	Timer 1 overflow Flag. Set by hardware on Timer/Counter overflow. cleared by hardware when processor vectors to interrupt routine.
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TF0	TCON.5	Timer 0 overflow Flag. Set by hardware on Timer/Counter overflow. cleared by hardware when processor vectors to interrupt routine.
TR0	TCON.4	Timer 0 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
IE1	TCON.3	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected.Cleared when interrupt processed.
IT1	TCON.2	Intenupt 1 Type control bit. Set/ cleared by software to specify falling edge/low level triggered external interrupts.
IE0	TCON.1	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected.Cleared when interrupt processed.
IT0	TCON.0	Intenupt 0 Type control bit. Set/ cleared by software to specify falling edge/low level triggered external interrupts.

SCON register

LSB

bit	7	6	5	4	3	2	1	0
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.
RI : Receive interrupt flag.

T2CON: Timer/Counter 2 Control register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$

TF2 : Timer 2 overflow flag. TF2 is set by a Timer 2 overflow happens and must be cleared by software. TF2 will not be set when either RCLK=1 or TCLK=1.

EXF2 : Timer 2 external flag. Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX(P1.1) pin and EXEN2=1. When Timer 2 interrupt is enabled, EXF2=1 will cause the CPU to vector the Timer 2 interrupt routine. EXF2 must be cleared by software.EXF2 does not cause an interrupt in up/down mode(DCEN=1).

RCLK : Receive clock flag. When set, cause the serial port to use Timer 2 overflow pulses for its receive clock in modes 1 and 3. When cleared, cause Timer 1 overflow to be used for the receive clock.

TCLK : Transmit clock flag.When set, cause the serial port to use Timer 2 overflow pulses for its transmit clock in modes 1 and 3. When cleared, cause Timer 1 overflows to be used for the transmit clock.

EXEN2 : Timer 2 external enable flag. When set, allows a capture or reload to occur as a result of a negative transition on T2EX(P1.1) pin if Timer 2 is not being used to clock the serial port. When cleared, cause Timer 2 to ignore events at T2EX(P1.1) pin.

TR2 : Timer 2 Run control bit.When set, start the Timer 2. When cleared, stop the Timer 2.

C/$\overline{T2}$: Timer or counter selector.

0: Select Timer 2 as internal timer function.

1: Select Timer 2 as external event counter (falling edge triggered).

CP/$\overline{RL2}$: Capture/Reload flag.

0 : Auto-reloads will occur either with Timer 2 overflows or negative transitions at T2EX pin when EXEN2=1.

1 : Captures will occur on negative transitions at T2EX pin if EXEN2=1.

XICON: Auxiliary Interrupt Control register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	PX3	EX3	IE3	IT3	PX2	EX2	IE2	IT2

PX3 : If set, Set priority for external interrupt 3 higher

EX3 : If set, Enables external interrupt 3.

IE3 : External Interrupt 3 Edge flag. Sets by hardware when external interrupt edge detected.Cleared by hardware when interrupt is starting to be serviced.

IT3 : External Interrupt 3 type control bit. Set/Cleared by software to specified falling edge/low level triggered interrupt.

PX2 : If set, Set priority for external interrupt 3 higher

EX2 : If set, enables external interrupt 2.

IE2 : External Interrupt 2 Edge flag. Sets by hardware when external interrupt edge detected. Cleared when interrupt is starting to be serviced.

IT2 : Interrupt 2 types control bit. Set/Cleared by software to specify falling edge/low level triggered interrupt.

IP (or XICON) and IPH are combined to form 4-level priority interrupt as the following table.

{IPH.x, IP.x/XICON.x}	Priority Level
1,1	0(highest)
1,0	1
0,1	2
0,0	3

6.3 Interrupt Priorities

Each interrupt source can also be individually programmed to one of four priority levels by setting or clearing a bit in Special Function Registers IP or XICON 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.	IE0	0(highest)
1.	TF0	1
2.	IE1	2
3.	TF1	3
4.	RI +Tl	4
5.	TF2+EXF2	5
6.	IE2	6
7.	IE3	7

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

6.4 How Interrupts Are Handled

Each interrupt flag is sampled at every system clock cycle. The samples are polled during the next system clock. If one of the flags was in a set condition at first cycle, the second cycle(polling cycle) 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 or IP 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 or IP, 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 rising edge of the third machine cycle, then in accordance with the above rules it will be vectored to during the 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
IE0	0003H
TF0	000BH
IE1	0013H
TF1	001BH
RI+TI	0023H
TF2+EXF2	002BH
IE2	0033H
IE3	003BH

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

There are four external interrupt sources in STC89xx MCU. The external sources can be programmed to be level-activated or transition-activated by clearing or setting bits IT0, IT1, IT2 or IT3 in Registers TCON or XICON. 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 $\overline{\text{INT0}}$ 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 $\overline{\text{INT0}}$, $\overline{\text{INT1}}$, $\overline{\text{INT2}}$ and $\overline{\text{INT3}}$ levels are inverted and latched into the interrupt flags IE0, IE1, IE2 and IE3 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.

Chapter 7 Timer/Counter

STC89xx has three 16-bit timers, and they are named T0, T1 and T2. Each of them can also be individually configured as timers or event counters.

In the “Timer” function, the register is incremented every 12 system clocks or 6 system clocks depending on 12T mode or 6T mode that the user configured this device.

In the “Counter” function, the register is incremented in response to a 1-to-0 transition at its corresponding external input pin, T0,T1 or T2. 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 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.

The “Timer” or “Counter” function of Timer 0 and Timer 1 is selected by control bits $C/\overline{T}$ in the Speciai Function Register TMOD. And the “Timer” or “Counter” function of Timer 2 is selected by control bit $C/\overline{T2}$ in the Speciai Function Register T2CON.

There are two SFR designed to configure timers T0 and T1. They are TMOD, TCON.
There are extra two SFR designed to configure timer T2. They are T2MOD, T2CON.

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
T2CON	Timer/Counter 2 control	C8H	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	$C/\overline{T2}$	$CP/\overline{RL2}$	0000 0000B
T2MOD	Timer/Counter 2 mode	C9H	-	-	-	-	-	-	T2OE	DCEN	xxxx xx00B
RCAP2L	Timer/Counter 2 Reload/ Capture High Byte	CAH									0000 0000B
RCAP2H	Timer/Counter 2 Reload/ Capture High Byte	CBH									0000 0000B
TL2	Timer/Counter 2 Low Byte	CCH									0000 0000B
TH2	Timer/Counter 2 High Byte	CDH									0000 0000B

TCON register: Timer/Counter Control Register

(MSB) (LSB)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Symbol	Position	Name and Significance	Symbol	Position	Name and Significance
TF1	TCON.7	Timer 1 overflow Flag. Set by hardware on Timer/Counter overflow. cleared by hardware when processor vectors to interrupt routine.	IE1	TCON.3	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected.Cleared when interrupt processed.
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter on/off.	IT1	TCON.2	Intenupt 1 Type control bit. Set/ cleared by software to specify falling edge/low level triggered external interrupts.
TF0	TCON.5	Timer 0 overflow Flag. Set by hardware on Timer/Counter overflow. cleared by hardware when processor vectors to interrupt routine.	IE0	TCON.1	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected.Cleared when interrupt processed.
TR0	TCON.4	Timer 0 Run control bit. Set/cleared by software to turn Timer/Counter on/off.	IT0	TCON.0	Intenupt 0 Type control bit. Set/ cleared by software to specify falling edge/low level triggered external interrupts.

TMOD register : Timer/Counter Mode Control Register

(MSB) (LSB)

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
Timer 1				Timer 0			

GATE — Gating control when set. Timer/Counter "x" is enabled only while "$\overline{INTx}$ " pin is high and "TRx"control pin is set.When cleared Timer "x" is enabled whenever "TRx" control bit is set.

C/$\overline{T}$ — Timer or Counter Selector cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin).

M0	M1	Operating Mode
0	0	B-bit Timer/Counter "THx" with "TLx" as 5-bit prescaler.
0	1	16-bit Timer/Counter"THx"and"TLx"are cascaded;there is no prescaler
1	0	8-bit auto-reload Timer/Counter “THx” holds a value which is to be reloaded into “TLx” each time it overflows.
1	1	(Timer 0) 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.
1	1	(Timer 1) Timer/Counter 1 stopped

T2CON: Timer/Counter 2 Control register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$

TF2 : Timer 2 overflow flag. TF2 is set by a Timer 2 overflow happens and must be cleared by software. TF2 will not be set when either RCLK=1 or TCLK=1.

EXF2 : Timer 2 external flag. Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX(P1.1) pin and EXEN2=1. When Timer 2 interrupt is enabled, EXF2=1 will cause the CPU to vector the Timer 2 interrupt routine. EXF2 must be cleared by software.EXF2 does not cause an interrupt in up/down mode(DCEN=1).

RCLK : Receive clock flag. When set, cause the serial port to use Timer 2 overflow pulses for its receive clock in modes 1 and 3. When cleared, cause Timer 1 overflow to be used for the receive clock.

TCLK : Transmit clock flag.When set, cause the serial port to use Timer 2 overflow pulses for its transmit clock in modes 1 and 3. When cleared, cause Timer 1 overflows to be used for the transmit clock.

EXEN2 : Timer 2 external enable flag. When set, allows a capture or reload to occur as a result of a negative transition on T2EX(P1.1) pin if Timer 2 is not being used to clock the serial port. When cleared, cause Timer 2 to ignore events at T2EX(P1.1) pin.

TR2 : Timer 2 Run control bit.When set, start the Timer 2. When cleared, stop the Timer 2.

C/$\overline{T2}$: Timer or counter selector.

0: Select Timer 2 as internal timer function.

1: Select Timer 2 as external event counter (falling edge triggered).

CP/$\overline{RL2}$: Capture/Reload flag.

0 : Auto-reloads will occur either with Timer 2 overflows or negative transitions at T2EX pin when EXEN2=1.

1 : Captures will occur on negative transitions at T2EX pin if EXEN2=1.

T2MOD: Timer/Counter 2 Mode register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	-	-	-	-	-	-	T20E	DCEN

T2OE : Timer 2 Output Enable bit. It enables Timer 2 overflow rate to toggle P1.0.

DCEN: Down Count Enable bit. When set, this allows Timer 2 to be configured as down counter

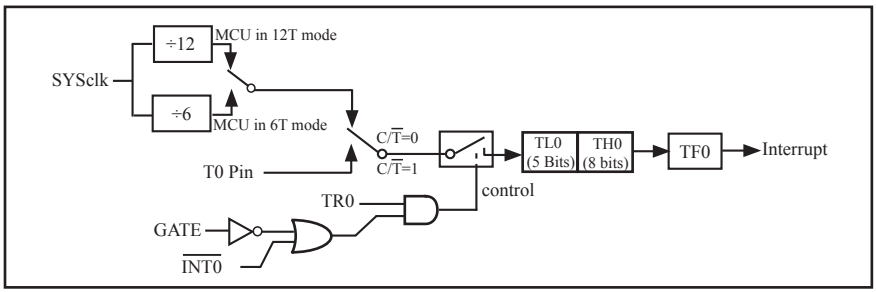
7.1 Timer/Counter 0 Mode of Operation

Mode 0

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 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).



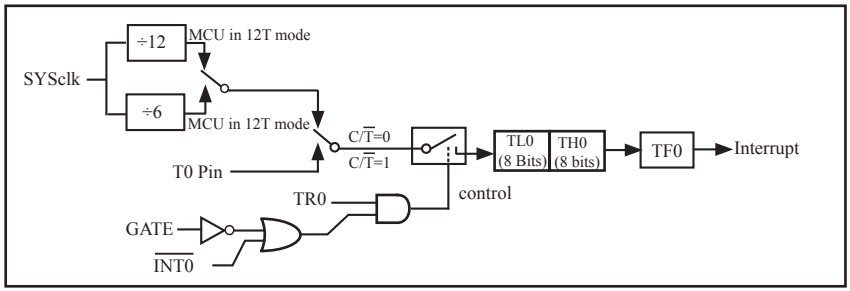
Timer/Counter 0 Mode 0: 13-Bit Counter

Mode 1

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.

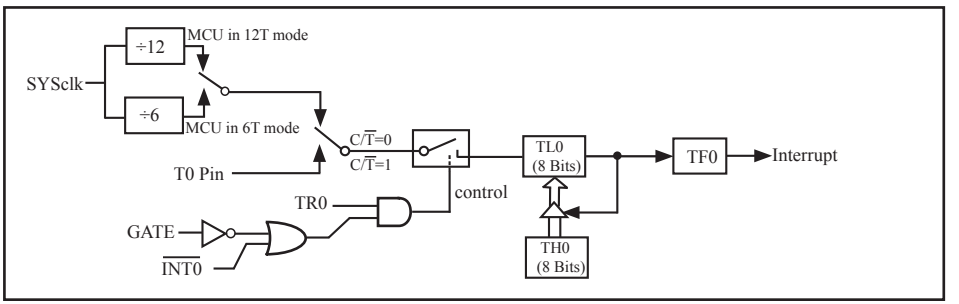
Mode 1 is the same as Mode 0, except that the timer register is being run with all 16 bits.



Timer/Counter 0 Mode 1 : 16-Bit Counter

Mode 2

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. Mode 2 operation is the same for Timer 0 and Timer 1.

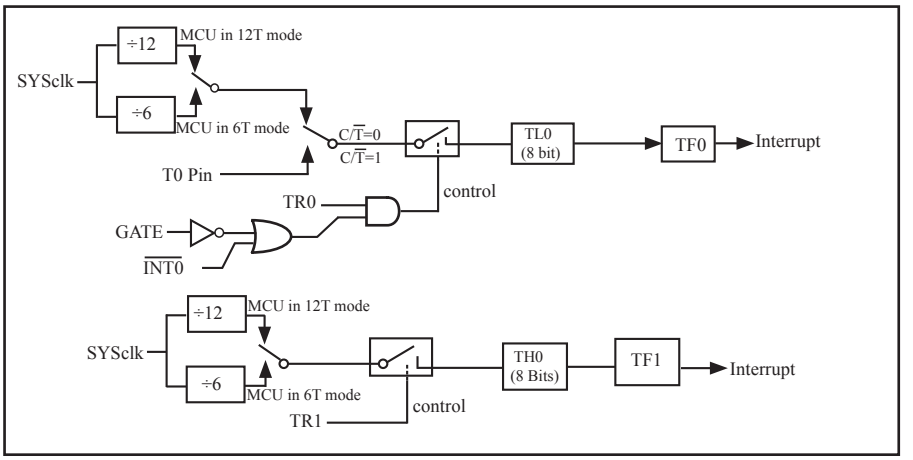


Timer/Counter 0 Mode 2: 8-Bit Auto-Reload

Mode 3

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 Counters

Example: write a program using Timer 0 to create a 5KHz square wave on P1.0.

Assembly Language Solution:

```
        ORG     0030H
        MOV     TMOD,   #20H            ;8-bit auto-reload mode
        MOV     TL0,    #9CH            ;initialize TL0
        MOV     TH0,    #9CH            ;-100 reload value in TH0
        SETB    TR0                     ;Start Tmier 0
LOOP:   JNB     TF0,    LOOP            ;Wait for overflow
        CLR     TF0                     ;Clear Timer overflow flag
        CPL     P1.0                    ;Toggle port bit
        SJMP    LOOP                    ;Repeat
        END
```

C Language Solution using Timer Interrupt :

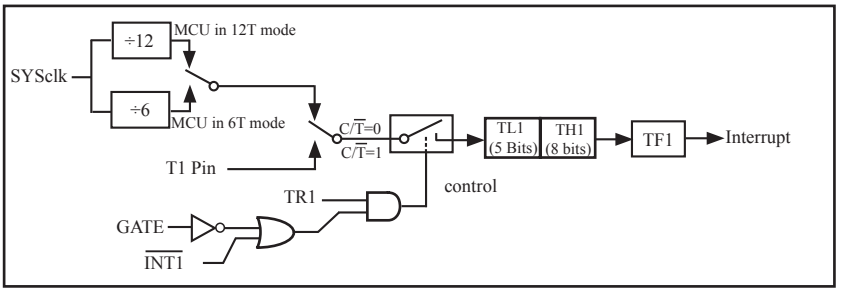
```
        #include <REG51.H>              /* SFR declarations */
        sbit    portbit = P1^0;         /* Use variable portbit to refer to P1.0 */
        main( )
        {
                TMOD = 0x02;            /* timer 0, mode 2 */
                TH0 = 9CH;              /* 100us delay */
                TR0 = 1;                /* Start timer */
                IE = 0x82               /* Enable timer 0 interrupt */
                while(1);               /* repeat forever */
        }

        void T0INT(void) interrupt 1
        {
                portbit = !portbit;     /*toggle port bit P1.0 */
        }
```

7.2 Timer/Counter 1 Mode of Operation

Mode 0

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.



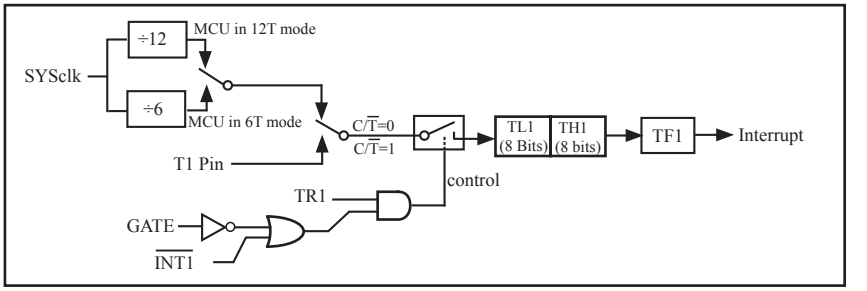
Timer/Counter 1 Mode 0: 13-Bit Counter

Mode 1

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 Counter

Example: write a program using Timer 1 to create a 500Hz square wave on P1.0.

Assembly Language Solution:

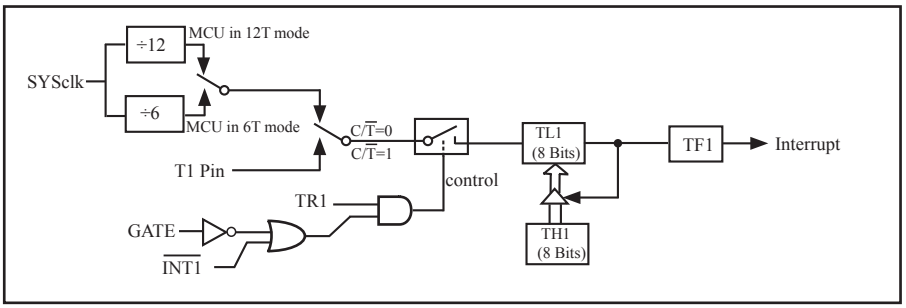
```
        ORG     0030H
        MOV     TMOD,   #10H            ;16-bit timer mode
        MOV     TH1,    #0F8H           ;-1000 (high byte)
        MOV     TH1,    #30H            ;-1000  (low byte)
        SETB    TR1                     ;Start Tmier 1
LOOP:   JNB     TF1,    LOOP            ;Wait for overflow
        CLR     TF1                     ;Clear Timer overflow flag
        CPL     P1.0                    ;Toggle port bit
        SJMP    LOOP                    ;Repeat
        END
```

C Language Solution :

```
#include <REG51.H>                      /* SFR declarations */
sbit    portbit = P1^0;                 /* Use variable portbit to refer to P1.0 */
main( )
{
        TMOD = 0x10;                    /* timer 1, mode 1, 16-bit timer mode */
        while (1)                       /* repeat forever */
        {
                TH1 = 0xF8;             /* -1000 (high byte) */
                TL1 = 0x30;             /* -1000 (low byte) */
                TR1 = 1;                /* Start timer 1 */
                while (TF0 !=1);        /* wait for overflow */
                TR1 = 0;                /* stop timer 1 */
                TF0 = 0;                /* clear timer overflow flag */
                portbit = !(portbit);   /* toggle P1.0 */
        }
}
```

Mode 2

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

The following program is a assembly language code that domestrates Timer 1 of STC89xx series MCU acted as baud rate generator.

```
;/*---------------------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ---------------------------------------------------------------*/
;/* --- STC 89xx Series MCU Timer 1 acted as Baud Rate Generoter 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 STC89xx series MCU SFR
            AUXR    EQU     8EH
;*/---------------------------------------------------------------------------------------------------*/
;Define baud rate auto-reload counter
;*************************************************************************
;The following Reload-Count and Baud is based on SYSclk =22.1184MHz, 1T mode, SMOD=1
;RELOAD_COUNT            EQU     0FFH            ;Baud=1,382,400 bps
;RELOAD_COUNT            EQU     0FEH            ;Baud=691,200 bps
;RELOAD_COUNT            EQU     0FDH            ;Baud=460,800 bps
;RELOAD_COUNT            EQU     0FCH            ;Baud=345,600 bps
;RELOAD_COUNT            EQU     0FBH            ;Baud=276,480 bps
;RELOAD_COUNT            EQU     0FAH            ;Baud=230,400 bps
;RELOAD_COUNT            EQU     0F4H            ;Baud=115,200 bps
;RELOAD_COUNT            EQU     0E8H            ;Baud=57,600 bps
;RELOAD_COUNT            EQU     0DCH            ;Baud=38,400 bps
;RELOAD_COUNT            EQU     0B8H            ;Baud=19,200 bps
;RELOAD_COUNT            EQU     70H             ;Baud=9,600 bps
;*************************************************************************
;The following Reload-Count and Baud is based on SYSclk =1.8432MHz, 1T mode, SMOD=1
;RELOAD_COUNT            EQU     0FFH            ;Baud=115,200 bps
;RELOAD_COUNT            EQU     0FEH            ;Baud=57,600 bps
;RELOAD_COUNT            EQU     0FDH            ;Baud=38,400 bps
;RELOAD_COUNT            EQU     0FCH            ;Baud=28,800 bps
;RELOAD_COUNT            EQU     0FAH            ;Baud=19,200 bps
;RELOAD_COUNT            EQU     0F4H            ;Baud=9,600 bps
;RELOAD_COUNT            EQU     0E8H            ;Baud=4,800 bps
;RELOAD_COUNT            EQU     0D0H            ;Baud=2,400 bps
;RELOAD_COUNT            EQU     0A0H            ;Baud=1,200 bps
;*************************************************************************
;The following Reload-Count and Baud is based on SYSclk =18.432MHz, 1T mode, SMOD=1
;RELOAD_COUNT            EQU     0FFH            ;Baud=1,152,000 bps
;RELOAD_COUNT            EQU     0FEH            ;Baud=576,000 bps
;RELOAD_COUNT            EQU     0FDH            ;Baud=288,000 bps
;RELOAD_COUNT            EQU     0FCH            ;Baud=144,000 bps
;RELOAD_COUNT            EQU     0F6H            ;Baud=115,200 bps
;RELOAD_COUNT            EQU     0ECH            ;Baud=57,600 bps
```

```
;RELOAD_COUNT    EQU     0E2H        ;Baud=38,400 bps
;RELOAD_COUNT    EQU     0D8H        ;Baud=28,800 bps
;RELOAD_COUNT    EQU     0C4H        ;Baud=19,200 bps
;RELOAD_COUNT    EQU     088H        ;Baud=9,600 bps
;*------------------------------------------------------------------------------
;The following Reload-Count and Baud is based on SYSclk =18.432MHz, 1T mode, SMOD=0
;RELOAD_COUNT    EQU     0FFH        ;Baud=576,000 bps
;RELOAD_COUNT    EQU     0FEH        ;Baud=288,000 bps
;RELOAD_COUNT    EQU     0FDH        ;Baud=144,000 bps
;RELOAD_COUNT    EQU     0FCH        ;Baud=115,200 bps
;RELOAD_COUNT    EQU     0F6H        ;Baud=57,600 bps
;RELOAD_COUNT    EQU     0ECH        ;Baud=38,400 bps
;RELOAD_COUNT    EQU     0E2H        ;Baud=28,800 bps
;RELOAD_COUNT    EQU     0D8H        ;Baud=19,200 bps
;RELOAD_COUNT    EQU     0C4H        ;Baud=96,000 bps
;RELOAD_COUNT    EQU     088H        ;Baud=4,800 bps
;*------------------------------------------------------------------------------
;The following Reload-Count and Baud is based on SYSclk =18.432MHz, 12T mode, SMOD=0
RELOAD_COUNT    EQU     0FBH        ;Baud=9,600 bps
;RELOAD_COUNT    EQU     0F6H        ;Baud=4,800 bps
;RELOAD_COUNT    EQU     0ECH        ;Baud=2,400 bps
;RELOAD_COUNT    EQU     0D8H        ;Baud=1,200 bps
;*------------------------------------------------------------------------------
;The following Reload-Count and Baud is based on SYSclk =18.432MHz, 12T mode, SMOD=1
RELOAD_COUNT    EQU     0FBH        ;Baud=19,200 bps
;RELOAD_COUNT    EQU     0F6H        ;Baud=9,600 bps
;RELOAD_COUNT    EQU     0ECH        ;Baud=4,800 bps
;RELOAD_COUNT    EQU     0D8H        ;Baud=2,400 bps
;RELOAD_COUNT    EQU     0B0H        ;Baud=1,200 bps
;*------------------------------------------------------------------------------
;The following Reload-Count and Baud is based on SYSclk =11.0592MHz, 12T mode, SMOD=0
;RELOAD_COUNT    EQU     0FFH        ;Baud=28,800 bps
;RELOAD_COUNT    EQU     0FEH        ;Baud=14,400 bps
;RELOAD_COUNT    EQU     0FDH        ;Baud=9,600 bps
;RELOAD_COUNT    EQU     0FAH        ;Baud=4,800 bps
;RELOAD_COUNT    EQU     0F4H        ;Baud=2,400 bpsS
;RELOAD_COUNT    EQU     0E8H        ;Baud=1,200 bps
;*------------------------------------------------------------------------------
;*------------------------------------------------------------------------------
;The following Reload-Count and Baud is based on SYSclk =11.0592MHz, 12T mode, SMOD=1
;RELOAD_COUNT    EQU     0FFH        ;Baud=57,600 bps
;RELOAD_COUNT    EQU     0FEH        ;Baud=28,800 bps
;RELOAD_COUNT    EQU     0FDH        ;Baud=14,400 bps
;RELOAD_COUNT    EQU     0FAH        ;Baud=9,600 bps
;RELOAD_COUNT    EQU     0F4H        ;Baud=4,800 bps
;RELOAD_COUNT    EQU     0E8H        ;Baud=2,400 bps
;RELOAD_COUNT    EQU     0D0H        ;Baud=1,200 bps
;*------------------------------------------------------------------------------
```

```
;Define LED indicator
        LED_MCU_START           EQU     P1.7        ;MCU operating LED indicator
;-------------------------------------------------------------------------------------------------------------
                ORG             0000H
                AJMP            MAIN
;-------------------------------------------------------------------------------------------------------------
                ORG             0023H
                AJMP            UART_Interrupt      ;Jump into RS232 UART-Interrupt service subroutine
                NOP
                NOP
;-------------------------------------------------------------------------------------------------------------
MAIN:
                MOV     SP,     #7FH                ;Set stack pointer
                CLR     LED_MCU_START               ;Turn on MCU operating LED indicator
                ACALL   Initial_UART                ;Initialize UART
                MOV     R0,     #30H                ;30H = the ASCII code of printable character '0'
                MOV     R2,     #10                 ;Send ten characters '0123456789'
LOOP:
                MOV     A,      R0
                ACALL   Send_One_Byte               ;Send one byte
                ; if Character-Display, display '0123456789'
                ;if Hexadecimal-Display, display '30 31 32 33 34 35 36 37 38 39'
                INC     R0
                DJNZ    R2,     LOOP
MAIN_WAIT:
                SJMP    MAIN_WAIT                           ;infinite circle
;-------------------------------------------------------------------------------------------------------------
UART_Interrupt:                                     ;UART-Interrupt service subroutine
                JB      RI,     Is_UART_Receive
                CLR     TI                          ;Clear serial port transmit interrupt flag
                RETI
Is_UART_Receive:
                CLR     RI
                PUSH    ACC
                MOV     A,      SBUF                ;acquire the received byte
                ACALL   Send_One_Byte               ;re-send the received byte
                POP     ACC
                RETI
;-------------------------------------------------------------------------------------------------------------
Initial_UART:                                       ;Initialize UART
;SCON Bit:  7        6       5       4       3       2       1       0
;          SM0/FE   SM1     SM2     REN     TB8     RB8     TI      RI
                MOV     SCON,   #50H                ;0101,0000   8-bit variable baud rate,no odd parity bit
                MOV     TMOD,   #21H                ;Use Timer 1 as 8 bit auto-reload counter
                MOV     TH1,    #RELOAD_COUNT       ;Set auto-reload count of Timer 1
                MOV     TL1,    #RELOAD_COUNT
;-------------------------------------------------------------------------------------------------------------
;               ORL     PCON,   #80H                        ;baud rate double
```

```
;-------------------------------------------------------------------------------------------------------------
;                       ORL     AUXR,   #01000000B             ;Use Timer 1 in 1T mode
                        ANL     AUXR,   #10111111B             ;Use Timer 1 in 12T mode
;-------------------------------------------------------------------------------------------------------------
                        SETB    R1                             ; Start up Timer 1
                        SETB    ES
                        SETB    EA
                        RET
;-------------------------------------------------------------------------------------------------------------
;Portal parameter: A= the byte to send
Send_One_Byte:                                                 ;Send one byte
                        CLR     ES
                        CLR     TI                             ;Clear serial port transmit interrupt flag
                        MOV     SBUF,   A
Wait_Send_Finish:
                        JNB     TI,     Wait_Send_Finish       ;Wait to finish send
                        CLR     TI
                        SETB    ES
                        RET
;-------------------------------------------------------------------------------------------------------------
                        END
```

7.3 Timer/Counter 2 Mode of Operation

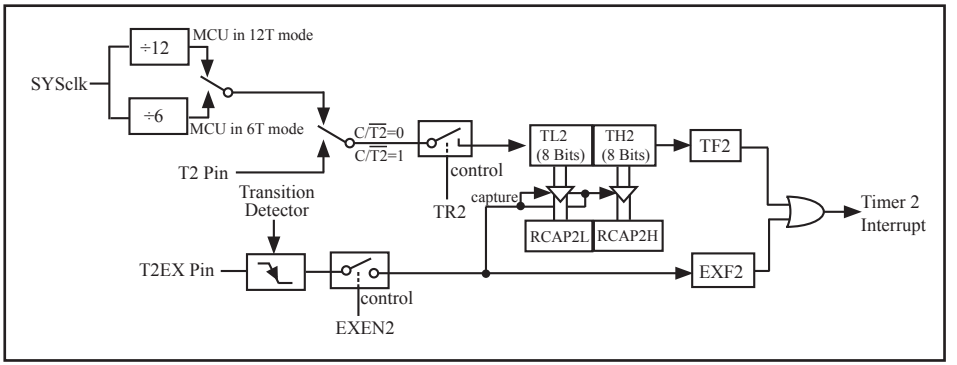
Timer 2 is a 16-bit timer/counter which can operate as either an event timer or an event counter as selected by C/$\overline{T2}$ in the special function register T2CON. Timer 2 has four operation modes: Capture Mode, Auto-Reload Mode (up or down counting), Baud-Rate Generator Mode and Programable Clock-Out Mode, which are selected by bits T2CON and T2MOD as shown in following table.

Timer 2 Operating Modes Table

RCLK+TCLK	CP/$\overline{RL2}$	TR2	Mode
0	0	1	16-bit auto-reload
0	1	1	16-bit capture
1	X	1	buad rate generator
X	X	0	(off)

Capture Mode

In the capture mode there are two options selected by bit EXEN2 in T2CON. If EXEN2=0, Timer 2 is a 16-bit timer or counter which upon overflowing sets bit TF2 (Timer 2 overflow flag). This bit can then be used to generate an interrupt (by enabling the Timer 2 interrupt bit in the IE register). If EXEN2=1, Timer 2 still does the above, but with the added feature that a 1-to-0 transition at external input T2EX causes the current value in the Timer 2 registers, TH2 and TL2, to be captured into registers RCAP2H and RCAP2L, respectively. In addition, the transition at T2EX causes bit EXF2 in T2CON to be set, and the EXF2 bit, like TF2, can generate an interrupt which vectors to the same location as Timer 2 overflow interrupt. TF2 and EXF2 is ORed to request the interrupt service. The capture mode is illustrated in following figure.

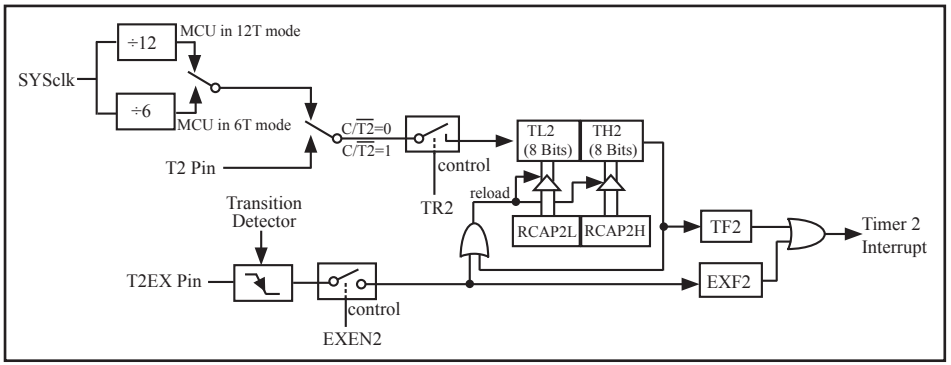


Timer 2 in Capture Mode

Auto-Reload Mode

In 16-bit auto-reload mode, Timer 2 can be configured to count up or down. The counting direction is determined by DCEN in special function register T2MOD and T2EX pin. If DCEN=0, counting up. If DCEN=1, the counting direction is determined by T2EX pin. If T2EX=1, counting up, otherwise counting down.

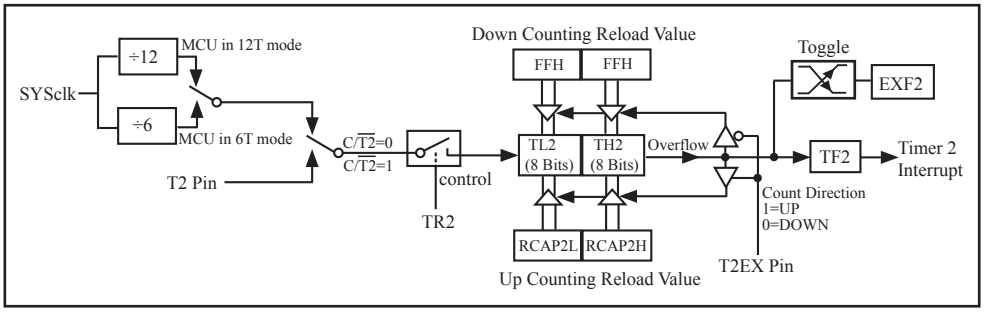
The following figure shows DCEN=0, which enables Timer 2 to count up automatically. In this mode there are two options selected by bit EXEN2 in T2CON register. If EXEN2=0, then Timer 2 counts up to 0FFFFH and sets the TF2 (Overflow Flag) bit upon overflow. This causes the Timer 2 registers to be reloaded with the 16-bit value in RCAP2L and RCAP2H. The values in RCAP2L and RCAP2H are preset by firmware. If EXEN2=1, then a 16-bit reload can be triggered either by an overflow or by a 1-to-0 transition at input T2EX. This transition also sets the EXF2 bit. The Timer 2 interrupt, if enabled, can be generated when either TF2 or EXF2 are 1.



Timer 2 in Auto-Reload Mode (DCEN=0)

The following figure shows DCEN=1, which enables Timer 2 to count up or down. This mode allows pin T2EX to control the counting direction. When a logic 1 is applied at pin T2EX, Timer 2 will count up. Timer 2 will overflow at 0FFFFH and set the TF2 flag, which can then generate an interrupt if the interrupt is enabled. This overflow also causes the 16-bit value in RCAP2L and RCAP2H to be reloaded into the timer registers TL2 and TH2. A logic 0 applied to pin T2EX causes Timer 2 to count down. The timer will underflow when TL2 and TH2 become equal to the value stored in RCAP2L and RCAP2H. This underflow sets the TF2 flag and causes 0FFFFH to be reloaded into the timer registers TL2 and TH2.

The external flag EXF2 toggles when Timer 2 underflows or overflows. This EXF2 bit can be used as a 17th bit of resolution if needed. The EXF2 flag does not generate an interrupt in this mode.



Timer 2 in Auto-Reload Mode (DCEN=1)

Buad-Rate Generator Mode

Timer2 can be configured to generate various baud-rate. Bit TCLK and/or RCLK in T2CON allow the serial port transmit and receive baud rates to be derived from either Timer1 or Timer2. When TCLK=0, Timer1 is used as the serial port transmit baud rate generator. When TCLK=1, Timer2 is used as the serial port transmit baud rate generator. RCLK has the same effect for the serial port baud rate. With these two bits, the serial port can have different receive and transmit baud rates – one generated from Timer 1 and the other from Timer 2.

In BRG mode, Timers is operated very like auto-reload up-only mode except that the T2EX pin cannot control reload. An overflow on Timer 2 will load RCAP2H, RCAP2L contents onto Timer2, but TF2 will not be set. A 1-to-0 transition on P2EX pin can set EXF2 to request interrupt service if EXEN2=1.

The following figure shows the Timer 2 in baud rate generation mode to generate RX Clock and TX Clock into UART engine. The baud rate generation mode is like the auto-reload mode, in that a rollover in TH2 causes the Timer 2 registers to be reloaded with the 16-bit value in registers RCAP2H and RCAP2L, which are preset by software.

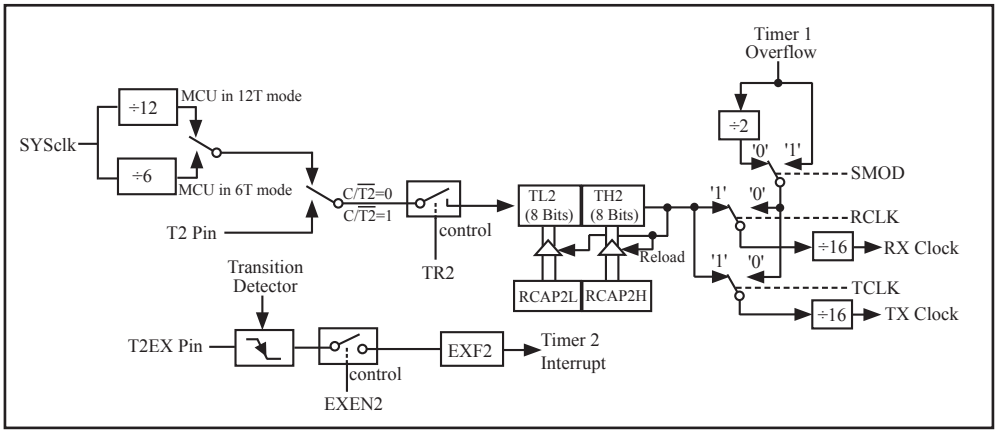
The baud rate in UART Mode 1 and Mode 3 are determined by Timer2's overflow rate given below:

$$\text{Baud Rate} = \frac{\text{Timer 2 overflow rate}}{16} \qquad \text{(counting T2EX)}$$

The Timer can be configured for either "timer" or "counter" operation. In the most typical applications, it is configured for "timer" operation($C/\overline{T2}=0$)."Timer" operation is a little different for Timer 2 when it's being used as a baud rate generator. Normally, as timer it would increment every machine clcye(thus at 1/6 or 1/12 the system clock).In that case the baud rate is given bu the formalu:

$$\text{Baud Rate} = \frac{\text{SYSclk}}{n\times[65536 - (\text{RCAP2H, RCAP2L})]} \qquad \text{(as timer)}$$

when MCU in 12T mode, n=32; When MCU in 6T mode, n=16.



Timer 2 in Baud-Rate Generator Mode

The Timer 2 as a baud rate generator mode is valid only if RCLK and/or TCLK=1 in T2CON register. Note that a rollover in TH2 does not set TF2, and will not generate an interrupt. Thus, the Timer 2 interrupt does not have to be disabled when Timer 2 is in the baud rate generator mode. Also if the EXEN2 (T2 external enable bit) is set, a 1-to-0 transition in T2EX (Timer/counter 2 trigger input) will set EXF2 (T2 external flag) but will not cause a reload from (RCAP2H, RCAP2L) to (TH2,TL2). Therefore when Timer 2 is in use as a baud rate generator, T2EX can be used as an additional external interrupt, if needed.

It should be noted that when Timer 2 is running (TR2=1) in "timer" function in the baud rate generator mode, one should not try to read or write TH2 and TL2. As a baud rate generator, Timer 2 is incremented at 1/2 the system clock or asynchronously from pin T2; under these conditions, a read or write of TH2 or TL2 may not be accurate. The RCAP2 registers may be read, but should not be written to, because a write might overlap a reload and cause write and/or reload errors. The timer should be turned off (clear TR2) before accessing the Timer 2 or RCAP2 registers.

The following program is an assembly language code that domestrates Timer 2 of STC89xx series MCU acted as baud rate generator.

```
;/*-------------------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ----------------------------------------------------------*/
;/* --- STC 89xx Series MCU Timer 2 as Baud Rate Generoter 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 STC89xx series MCU SFRs
;SFRs associated with Timer 2 and RS232 port
        T2CON           EQU     0C8H
        TR2             EQU     T2CON.2                 ;TR2 is the second bit in SFR—T2CON
```

```
        RCAP2L          EQU     0CAH
        RCAP2H          EQU     0CBH
        TH2             EQU     0CDH
        TL2             EQU     0CCH
;-------------------------------------------------------------------------------------------------------------
;Set baud rate auto-relaod value
        RELOAD_COUNT_HIGH   EQU     0FFH

;RELOAD_COUNT_HIGH must be set for 0FFH when the following parameters are used
;RELOAD_COUNT_LOW           EQU     0FAH            ;SYSclk=22.1184MHz, Baud=115200
;RELOAD_COUNT_LOW           EQU     0EEH            ;SYSclk=22.1184MHz, Baud=38400
;RELOAD_COUNT_LOW           EQU     0F0H            ;SYSclk=20.000MHz, Baud=38400
;RELOAD_COUNT_LOW           EQU     0F6H            ;SYSclk=12.000MHz, Baud=38400
;RELOAD_COUNT_LOW           EQU     0FDH            ;SYSclk=11.059MHz, Baud=115200
;RELOAD_COUNT_LOW           EQU     0F7H            ;SYSclk=11.059MHz, Baud=38400
;RELOAD_COUNT_LOW           EQU     0F8H            ;SYSclk=10.000MHz, Baud=38400
;RELOAD_COUNT_LOW           EQU     0FBH            ;SYSclk=6.000MHz, Baud=38400
;RELOAD_COUNT_LOW           EQU     0FCH            ;SYSclk=5.000MHz, Baud=38400
;RELOAD_COUNT_LOW           EQU     070H            ;SYSclk=11.059MHz, Baud=2400
;-------------------------------------------------------------------------------------------------------------
;calculate auto-reload value
;-------------------------------------------------------------------------------------------------------------
;RELOAD=INT(SYSclk/Baud/32+0.5), INT means geting integer part and abandoning fractional part
;change the decimal RELOAD to hexadecimal, and save the value from 1000H subtracting RELOAD into
RCAP2H, RCAP2L
;-------------------------------------------------------------------------------------------------------------
                ORG     0000H
                AJMP    MAIN
;-------------------------------------------------------------------------------------------------------------
                ORG     0023H                   ;RS232 serial port interrupt
                AJMP    UART
                NOP
                NOP
;-------------------------------------------------------------------------------------------------------------
MAIN:
                MOV     SP,     #0E0H
                ACALL   Initial_UART            ;Initialize serial port
                MOV     R0,     #30H            ;Send ten characters "0123456789"
                MOV     R2,     #10
LOOP:
                MOV     A,      R0
                ACALL   Send_One_Byte           ;Send one byte
                INC     R0
                DJNZ    R2,     LOOP
WAIT1:
                SJMP    WAIT1                   ;dynamic stop
```

```
;-------------------------------------------------------------------------------
UART:                                                   ;Serial port interrupt service routine
                JBC     RI,     UART_1
                RETI                                    ;Inquire when transmitting
UART_1:                                                 ;Receive one byte
                PUSH    ACC
                MOV     A,      SBUF                    ;Get the received byte
                ACALL   Send_One_Byte                   ;Send back the received byte
                POP     ACC
                RETI
;-------------------------------------------------------------------------------
Initial_UART:                                   ;Initialize serial port
;BIT        7       6       5       4       3       2       1       0
;SCON   SM0/FE  SM1     SM2     REN     TB8     RB8     TI      RI
                MOV     SCON,   #50H            ;0101,0000 8-bit variable baud rate, no parity
Init_RS232:
                MOV     A       #RELOAD_COUNT_HIGH          ;Baud auto-reload value
                MOV     RCAP2H,A
                MOV     TH2,    A
                MOV     A,      #RELOAD_COUNT_LOW
                MOV     RCAP2L,A
                MOV     TL2,    A
                MOV     T2CON,  #0x34                       ;Timer 2 as baud rate generator
                SETB    ES                                  ;Enable serial port interrupt
                SETB    EA
                RET
;-------------------------------------------------------------------------------
;Send_One_Byte:                                             ;Send one byte
                CLR     ES
                CLR     TI                                  ;clear UART transmit interrupt flag
                MOV     SBUF,   A

WAIT2:
                JNB     TI,     WAIT2                       ;Wait to finish transmitting
                CLR     TI
                SETB    ES
                RET
;-------------------------------------------------------------------------------
                END
;-------------------------------------------------------------------------------
```

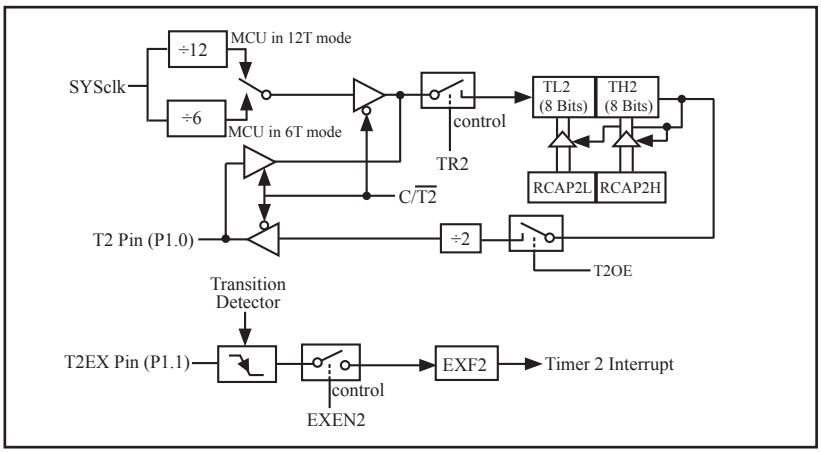
Programmable Clock Output Mode

The STC89xx seires is able to generate a programmable clock output on P1.0. When T2OE bit is set and C//T2 bit is cleared, Timer 2 overflow pulse will generate a 50% duty clock and output that to P1.0. The frequency of clock-out is calculated according to the following formula.

$$\text{Baud Rate} = \frac{\text{SYSclk}}{n \times [65536 - (\text{RCAP2H, RCAP2L})]}$$

when MCU in 12T mode, n=4; when MCU in 6T mode, n=2.
Note Timer 2 overflag, TF2 will always not be set in this mode.

The input clock, SYSclk/2, increments the 16-bit timer (TH2, TL2). The timer repeatedly counts to overflow from a loaded value. Once overflows occur, the contents of (RCAP2H, RCAP2L) are loaded into (TH2, TL2) for the consecutive counting. In the clock-out mode, Timer2 rollovers will not generate an interrupt. This is similar to when Timer 2 is used as a baud-rate generator. It is possible to use Timer 2 as a baud rate generator and a clock generator simultaneously. Note, however, that the baud-rate and the clock-out frequency depend on the same overflow rate of Timer 2.The following figure shows the Timer 2 in programmable clock output mode.



Timer 2 in Programmable Clock Output Mode

If Timer 2 in Programmabel Clock Out mode, some operations as shown below should be done:

- Set T2OE bit in T2MOD register.
- Clear C/T2 bit in T2CON register.
- Determine the 16-bit reload value from the formula and enter it in the RCAP2H and RCAP2L registers.
- Enter the same reload value as the initial value in the TH2 and TL2 registers.
- Set TR2 bit in T2CON register to start the Timer 2.

The following program is an assembly language code that domestrates Timer 2 programable clock/pulse out function on P1.0.

```
;/*-------------------------------------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ----------------------------------------------------------------*/
;/* --- STC 89xx Series MCU Timer 2 in programmable clock/pulse output mode,on P1.0------------*/
;/* If you want to use the program or the program referenced in the  */
;/* article, please specify in which data and procedures from STC    */
;/*-------------------------------------------------------------------------------------------------------*/
;Declare STC89xx series MCU SFRs
;SFRs associated with Timer 2 and RS232 port
        T2CON           EQU     0C8H
        T2MOD           EQU     0C9H
        TR2             EQU     T2CON.2                 ;TR2 is the second bit in SFR—T2CON
        RCAP2L          EQU     0CAH
        RCAP2H          EQU     0CBH
        TH2             EQU     0CDH
        TL2             EQU     0CCH
;Timer/Counter 2 control register T2CON
;               B7      B6      B5      B4      B3      B2      B1      B0      Reset Value
;bit-address    CF      CE      CD      CC      CB      CA      C9      C8
;T2CON(C8H)     TF2     EXF2    RCLK    TCLK    EXEN2   TR2     C//T2   CP//RL2     00
;T2MOD Register
;               B7      B6      B5      B4      B3      B2      B1      B0      Reset Value
;T2CON(C9H)     -       -       -       -       -       -       T2OE    DCEN    xxxxxx00B
;-------------------------------------------------------------------------------------------------------
        ORG     0000H
        AJMP    MAIN
;-------------------------------------------------------------------------------------------------------
        ORG     0100H
MAIN:
        MOV     SP,     #0E0H
        MOV     P1,     #0FFH                   ;turn off P1 port LED
        ACALL   SET_T2_OUT_MODE                 ;Set Timer 2 as high-speed pulse output mode
        MOV     DPTR,   #0FFF0H                 ;Set Timer pulse output velocity
        ACALL   SET_T2_OUT_SPEED
WAIT1:
        SJMP    WAIT1

        ACALL   DELAY
        ACALL   PAUSE                           ;Pause in order to observation
        MOV     DPTR,   #0FFE0H                 ;Set Timer pulse output velocity,lower 1/2 than last
        ACALL   SET_T2_OUT_SPEED
        ACALL   DELAY
        ACALL   PAUSE                           ;Pause in order to observation
        MOV     DPTR    #0FFD0H                 ;Set Timer pulse output velocity,lower 1/3 than last
```

```
                ACALL   SET_T2_OUT_SPEED
                ACALL   DELAY
                ACALL   PAUSE                           ;Pause in order to observation

WAIT2:
                SJMP    WAIT2                           ;dynamic stop
;-------------------------------------------------------------------------

DELAY:
                MOV     R1,     #0
                MOV     R2,     #0
                MOV     R3,     #30
DELAY_LOOP:
                DJNZ    R1,     DELAY_LOOP
                DJNZ    R2,     DELAY_LOOP
                DJNZ    R3,     DELALY_LOOP
                RET
;-------------------------------------------------------------------------
SET_T2_OUT_MODE:                                        ;Set Timer2 as pulse output mode
                MOV     T2CON,  #0                      ;Set Timer 2 as "timer" mode
                MOV     T2MOD, #02                      ;0000,0010, enable
                                                        ;Timer 2 overflow pulse outputting on P1.0
;-------------------------------------------------------------------------
SET_T2_OUT_SPEED:                                       ;Set Timer 2 pulse output velocity
                CLR     TR2                             ;Disable Timer 2
                MOV     RCAP2H,         DPH
                MOV     RCAP2L,         DPL
                SETB    TR2                             ;Triggle Timer 2
                RET
;-------------------------------------------------------------------------
PAUSE:                                                  ;Close Timer 2 function
                CLR     TR2
                MOV     P1,     #0FFH                   ;turn off P1 port LED
                ACALL   DELAY
                RET
;-------------------------------------------------------------------------
                END
;-------------------------------------------------------------------------
```

Chapter 8 UART with enhanced function

The serial port of STC89xx series 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 registers in the chip, one is for transmit and the other is for receive.

The serial port 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.

Symbol	**Description**	**Address**	Bit Address and Symbol MSB LSB								Value after Power-on or Reset
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	-	POF	GF1	GF0	PD	IDL	00x1 0000B
IE	Interrupt Enable	A8H	EA	-	ET2	ES	ET1	EX1	ET0	EX0	0000 0000B
IP	Interrupt Priority Low	B8H	-	-	PT2	PS	PT1	PX1	PT0	PX0	xx00 0000B
IPH	Interrupt Priority High	B7H	PX3H	PX2H	PT2H	PSH	PT1H	PX1H	PT0H	PX0H	0000 0000B
SADEN	Slave Address Mask	B9H									0000 0000B
SADDR	Slave Address	A9H									0000 0000B

SCON register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
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.

RI : Receive interrupt flag.

SBUF register

LSB

bit	7	6	5	4	3	2	1	0
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.

PCON: Power Control register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	SMOD	SMOD0	-	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.

IE: Interrupt Enable Rsgister

(MSB) (LSB)

EA	-	ET2	ES	ET1	EX1	ET0	EX0

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

Symbol	Position	Function
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.
ES	IE.4	Serial Port interrupt enable bit

IP: Interrupt Priority Low Register

(MSB) (LSB)

-	-	PT2	PS	PT1	PX1	PT0	PX0

Priority bit = 1 assigns high priority .
Priority bit = 0 assigns low priority.

Symbol	Position	Function
PS	IP.4	Serial Port interrupt priority bit.

IPH: Interrupt Priority High Register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	PX3H	PX2H	PT2H	PSH	PT1H	PX1H	PT0H	PX0H

PSH : If set, Set priority for serial port highest

SADEN: Slave Address Mask register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name								

SADDR: Slave Address register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name								

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.

8.1 UART Mode of Operation

8-Bit Shift Register (Mode 0)

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, and 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. The terms "RxD" and "TxD" are misleading in this mode. The RxD line is used for both data input and output, and the TxD line serves as the clock.

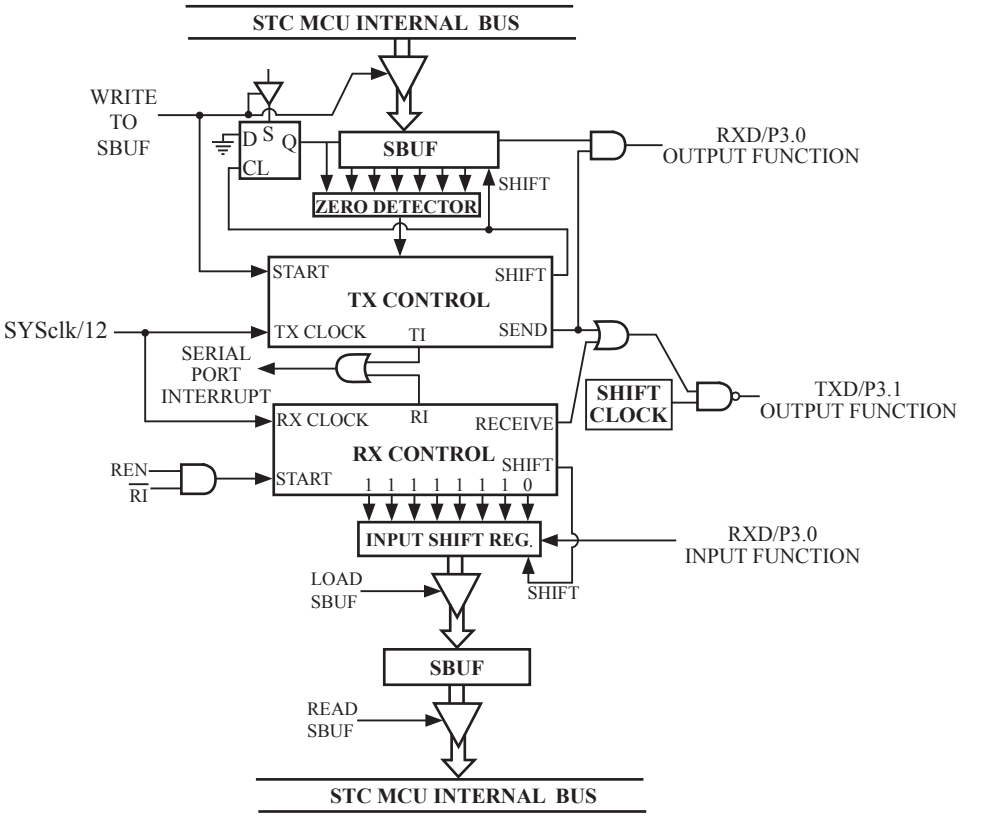
Transmission is initiated by any instruction that uses SBUF as a destination register. The “write to SBUF” signal also loads a “1” into the 9th position of the transmit shift register and tells the TX Control block to commence a transmission. The internal timing is such that one full system clock cycle will elapse between "write to SBUF," and activation of SEND.

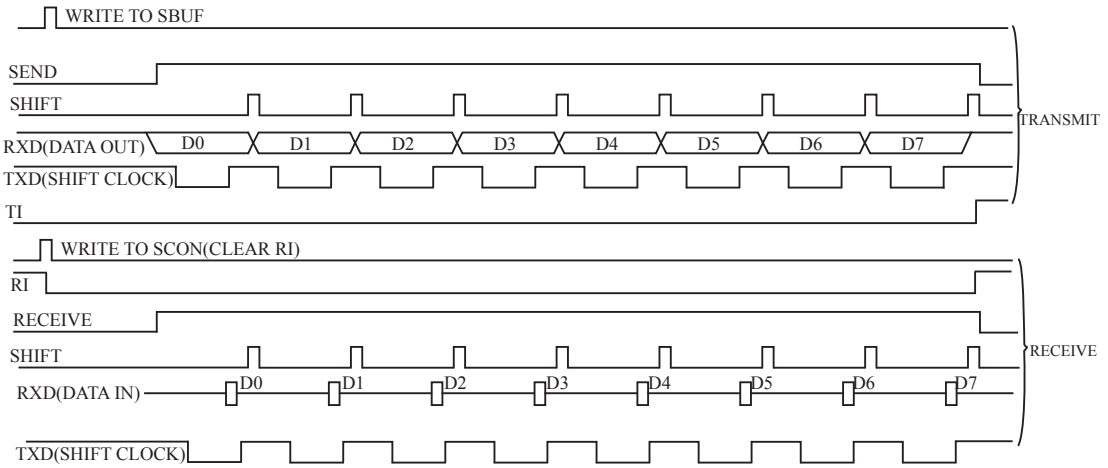
SEND transfers the output of the shift register to the alternate output function line of P3.0, and also transfers Shift Clock to the alternate output function line of P3.1. At the falling edge of the Shift Clock, the contents of the shift register are shifted one position to the right.

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-Bit UART with Variable Baud Rate (Mode 1)

In mode 1 the STC89xx serial port operates as an 8-bit UART with variable baud rate. A UART, or "universal asynchronous receiver/transmitter," is a device that receives and transmits serial data with each data character preceded by a start bit(low) and followed by a stop bit(high). A parity bit is sometimes inserted between the last data bit and the stop bit. The essential operation of a UART is parallel-to-serial conversion of output data and serial-to-parallel conversion of input data.

In mode 1, 10 bits are transmitted through TXD or received through RXD. The frame data includes a start bit (always 0), 8 data bits (LSB first) and a stop bit (always 1). For a receive operation, the stop bit goes into RB8 in SFR – SCON. The baud rate is determined by the overflow rate of Timer 1 or Timer 2 .

Baud rate in mode 1 = $(2^{SMOD}/32)$ x timer 1 overflow rate
or = $(2^{SMOD}/16)$ x Timer 2 overflow rate

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.

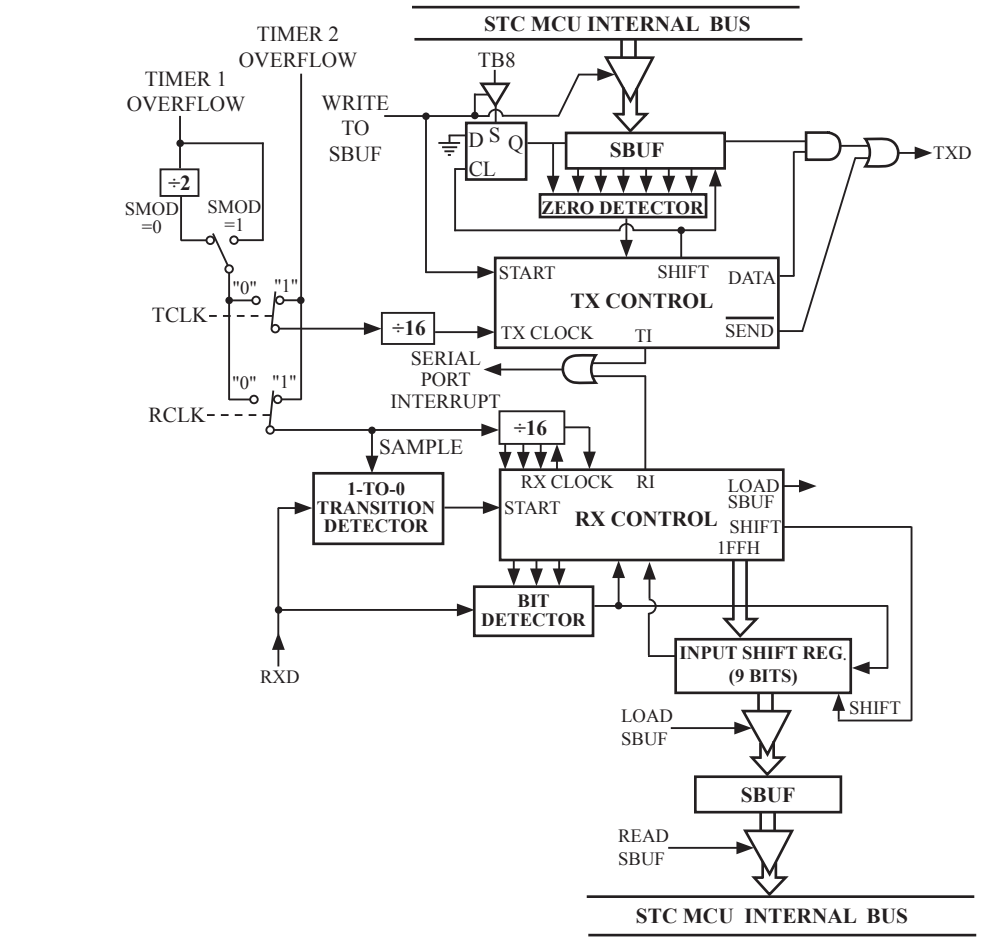
The transmission begins with activation of $\overline{SEND}$, which puts the start bit at TXD. One bit time later, DATA is activated, which enables the output bit of the transmit shift register to TXD. The first shift pulse occurs one bit time after that.

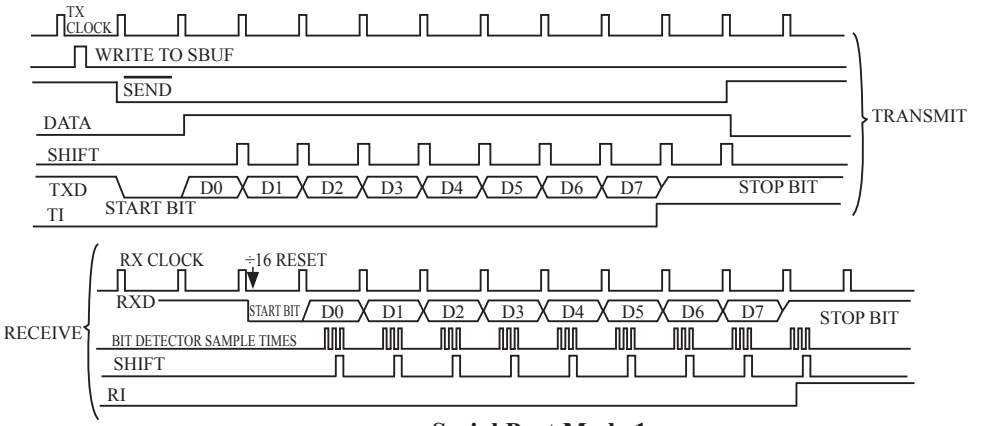
As data bits shift out to the right, zeroes are clocked in from the left. When the MSB of the data byte is at the output position of the shift register, then the 1 that was initially loaded into the 9th position is just to the left of the MSB, and all positions to the left of that contain zeroes. This 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 SM2=0 and 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

9-Bit UART with Fixed Baud Rate (Mode 2)

When SM1=1 and SM0=0, the serial port operates in mode 2 as a 9-bit UART with a 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 9^{th} 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 9^{th} 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 11^{th} 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.

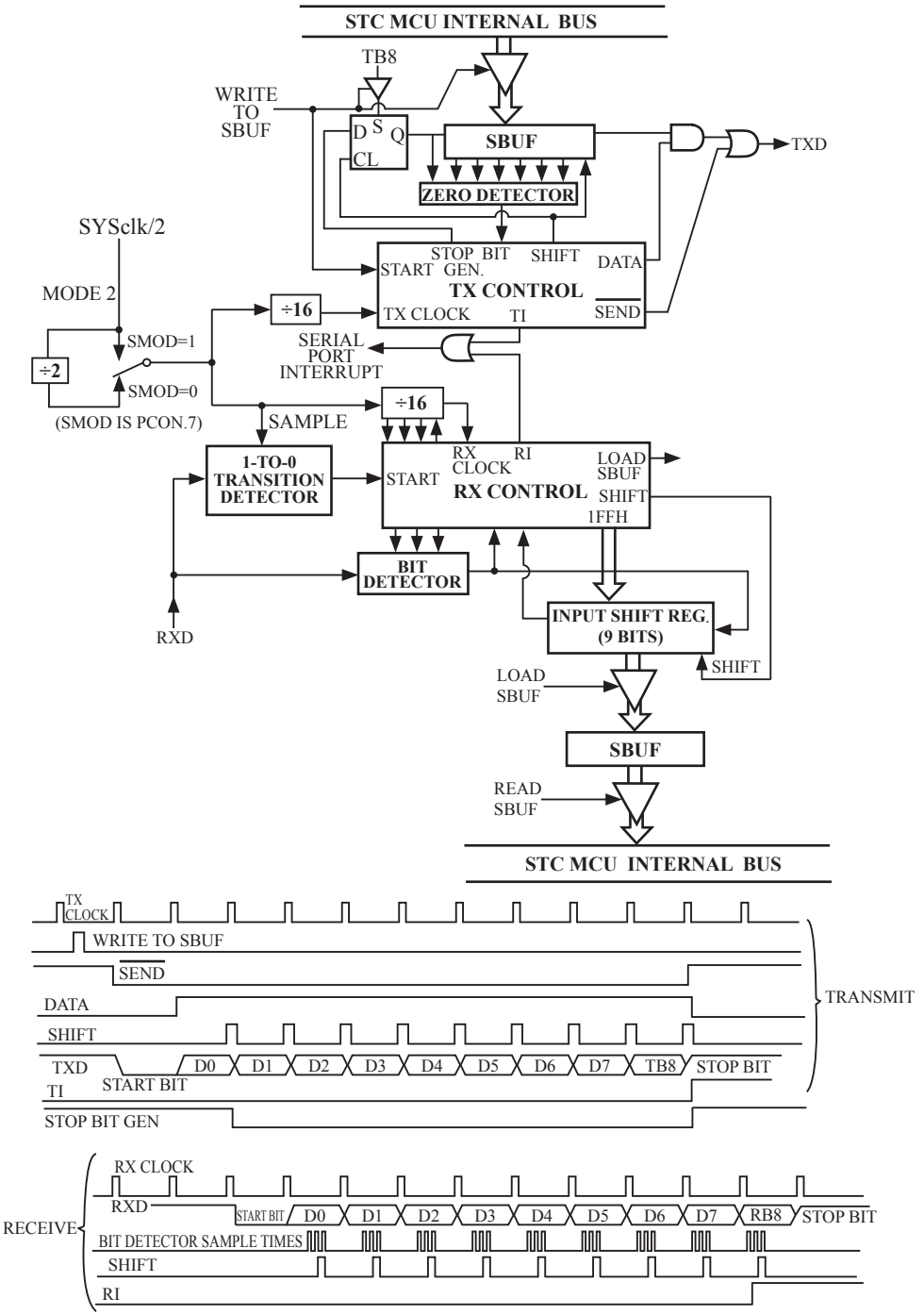
At the 7^{th}, 8^{th} and 9^{th} counter states of each bit time, the bit detector samples the value of RXD. The value accepted is the value that was seen in at least 2 of the 3 samples. This is done to reject noise. In order to reject false bits, if the value accepted during the first bit time is not 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 9^{th} 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

9-Bit UART with Variable Baud Rate (Mode 3)

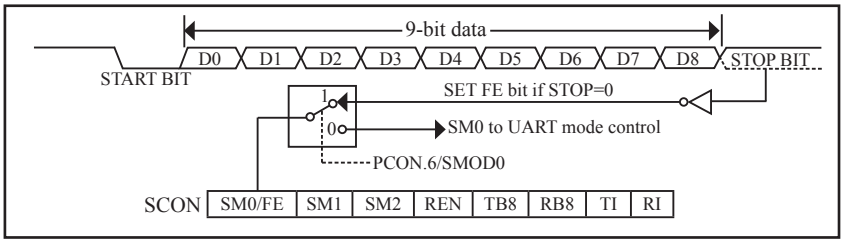
Mode 3, 9-bit UART with variable baud rate, is the same as mode 2 except the baud rate is variable.

$$\text{Baud rate in mode 3} = (2^{SMOD}/32) \times \text{Timer 1overflow rate}$$
$$\text{or} = (2^{SMOD}/16) \times \text{Timer 2 overflow rate}$$

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.

8.2 Frame Error Detection

When used for frame error detect, the UART looks for missing stop bits in the communication. A missing bit in stop 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.



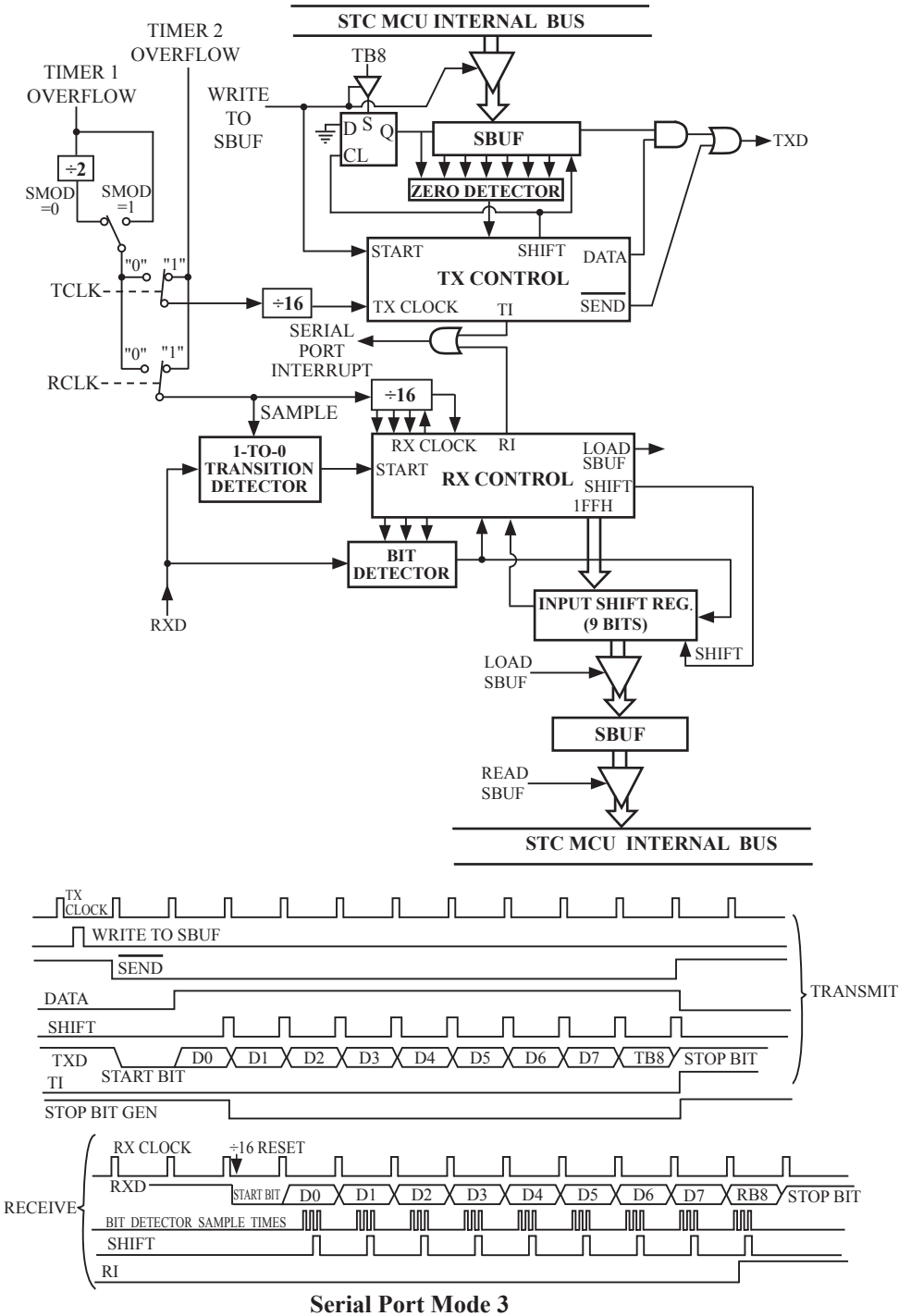
UART Frame Error Detection

8.3 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.



Serial Port Mode 3

8.4 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 STC89xx 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
        CLR     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
```

8.5 Buad Rates

The baud rate in Mode 0 is fixed:

$$\text{Mode 0 Baud Rate} = \frac{\text{SYSclk}}{12}$$

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 STC89xx series, the baud rates in Modes 1 and 3 are determined by Timer1 or Timer 2 overflow rate. The baud rate in Mode 1 and 3 are fixed:

Mode 1,3 Baud rate = $(2^{SMOD}/32)$ x timer 1 overflow rate
= $(2^{SMOD}/32)$ x timer 2 overflow rate

Timer 1 overflow rate = (SYSclk/12)/(256 - TH1);
Timer 2 overflow rate = SYSclk/(65536-(RCAP2H,RCAP2L))

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 table 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

When Timer 2 is used as the baud rate generator (either TCLK or RCLK in T2CON is '1'), the baud rate is as follows,

$$\text{Mode 1,3 Baud rate} = \frac{2^{SMOD} \times \text{SYSclk}}{32 \times (65536 - (\text{RCAP2H,RCAP2L}))}$$

The following table lists various commonly used baud rates generated by Timer 2.

Baud Rate		System Clocks /MHz	Timer 2	
12T mode	6T mode		RCAP2H	RCAP2L
375 000	750 000	12	FF	FF
9 600	19 200	12	FF	D9
2 800	9 600	12	FF	B2
2 400	4 800	12	FF	64
1 200	2 400	12	FE	C8
300	600	12	FB	1E
110	220	12	F2	AF
300	600	6	FD	8F
110	220	6	F9	57

Chapter 9 IAP / EEPROM

The ISP in STC89xx 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 90(max) pages. Each page contains 512 bytes. Dealing with flash, the user must erase it in page unit before writting (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 STC89xx 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 and option registers.

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.

9.1 IAP / ISP Control Register

The following special function registers are related to the IAP/ISP 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	-	-	-	-	-	MS2	MS1	MS0	xxxx x000B
ISP_TRIG	ISP/IAP Flash Command Trigger	E6H									xxxx xxxxB
ISP_CONTR	ISP/IAP Control Register	E7H	ISPEN	SWBS	SWRST	-	-	WT2	WT1	WT0	000x x000B
PCON	Power Control	87H	SMOD	SMOD0	-	POF	GF1	GF0	PD	IDL	00x1 0000B

ISP_DATA: ISP/IAP Flash Data Register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name								

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.

ISP_ADDRH: ISP/IAP Flash Address High

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name								

ISP_ADDRH 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: ISP/IAP Flash Address Low

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name								

ISP_ADDRL is the low port for all ISP/IAP modes. In page erase operation, it is ignored.

ISP_CMD: ISP/IAP Flash Command Register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	-	-	-	-	-	MS2	MS1	MS0

B7~B2: Reserved.
MS2, 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, 0 : Standby
0, 0, 1 : Data Flash/EEPROM read.
0, 1, 0 : Data Flash/EEPROM program.
0, 1, 1 : Data Flash/EEPROM page erase.

ISP_TRIG: ISP/IAP Flash Command Trigger.

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name								

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.

ISP_CONTR: ISP/IAP Control Register

LSB

bit	B7	B6	B5	B4	B3	B2	B1	B0
name	ISPEN	SWBS	SWRST	-	-	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.

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.

Note: Software reset actions could reset other SFR, but it never influence bits ISPEN and SWBS. The ISPEN and SWBS only will reset by power-up action, while not software reset.

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	Program <=55uS	Sector Erase <=21mS	Recommended System Clock Frequency (MHz)
0	1	1	6 SYSclks	30 SYSclks	5471 SYSclks	5MHz
0	1	0	11 SYSclks	60 SYSclks	10942 SYSclks	10MHz
0	0	1	22 SYSclks	120SYSclks	21885SYSclks	20MHz
0	0	0	43 SYSclks	240 SYSclks	43769 SYSclks	40MHz

9.2 STC89xx series internal EEPROM Selection Table

STC89xx series MCU internal EEPROM Selection Table				
Type	**EEPROM (Byte)**	**Sector Numbers**	**Begin_Sector Begin_Address**	**End_Sector End_Address**
STC89C/LE51RC	4K	8	0000H	0FFFH
STC89C/LE52RC	4K	8	0000H	0FFFH
STC89C/LE54RD+	45K	90	0000H	0B3FFH
STC89C/LE58RD+	29K	58	0000H	73FFH

9.3 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,      #0      ;Set address high(00h), Data ptr point to non-EEPROM area
        ;MOV    ISP_ADDRL,      #0      ;Clear IAP address to prevent misuse
        SETB    EA                      ;Set global enable bit

;/*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,      #0      ;Set address high(00h), Data ptr point to non-EEPROM area
        ;MOV    ISP_ADDRL,      #0      ;Clear IAP address to prevent misuse
        SETB    EA                      ;Set global enable bit
;/*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,      #0      ;Set address high(00h), Data ptr point to non-EEPROM area
        ;MOV    ISP_ADDRL,      #0      ;Clear IAP address to prevent misuse
```

9.4 Operating internal EEPROM Demo by Assembly Language

```
;/*---------------------------------------------------------------------------*/
;/* --- STC MCU International Limited ------------------------------------*/
;/* --- STC89xx 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 */
ISP_DATA        EQU     0E2H            ;Flash data register
ISP_ADDRH       EQU     0E3H            ;Flash address HIGH
ISP_ADDRL       EQU     0E4H            ;Flash address LOW
ISP_CMD         EQU     0E5H            ;Flash command register
ISP_TRIG        EQU     0E6H            ;Flash command trigger
ISP_CONTR       EQU     0E7H            ;Flash control register

;/*Define ISP/IAP/EEPROM operation const for IAP_CONTR*/
ENABLE_ISP      EQU     80H             ;if SYSCLK>20MHz
;ENABLE_ISP     EQU     81H             ;if SYSCLK<20MHz
ENABLE_ISP      EQU     82H             ;if SYSCLK<10MHz
;ENABLE_ISP     EQU     83H             ;if SYSCLK<5MHz
DEBUG_DATA      EQU     5AH

;Select MCU type
DATA_FLASH_START_ADDRESS    EQU     2000H               ;STC89C/LE52RC
;-------------------------------------------------
            ORG     0000H
            LJMP    MAIN
;-------------------------------------------------
            ORG     0100H
MAIN:
            MOV     P1,     #0F0H           ;1111,0000 System Reset OK
            LCALL   DELAY                   ;Delay
            MOV     P1,     #0F0H           ;1111,0000 System Reset OK
            LCALL   DELAY                   ;Delay
            MOV     SP,     #0E0H           ;Initialize stack pointer
;**********************************
;Read the first byte written into Flash
MAIN1:
            MOV     DPTR,   #DATA_FLASH_START_ADDRESS
            LCALL   Byte_Read
            MOV     40H,    A
            CJNE    A,      #DEBUG_DATA,    DATA_NOT_EQU_DEBUG_DATA
```

```
DATA_IS_DEBUG_DATA:
                MOV     P1,     #0111111B       ;(DATA_FLASH_START_ADDRESS) = #5A,
                                                ;turn on P1.7 LED
                LCALL   Delay                   ;delay
                MOV     A,      40H
                CPL     A
                MOV     P1,     A               ;Data is right and showed in port 1
WAIT1:
                SJMP    WAIT                    ;Stop
DATA_NOT_EQU_DEBUG_DATA:
                MOV     P1,     #11110111B      ;(DATA_FLASH_START_ADDRESS) != #5A,
                                                ;turn on P1.3 LED
                LCALL   Delay                   ;delay
                MOV     A,      40H
                CPL     A
                MOV     P1,     A               ;Data is error and showed in port 1
                LCALL   Delay                   ;delay

                MOV     DPTR,   #DATA_FLASH_START_ADDRESS
                ACALL   Sector_Erase            ;Sector erase,
                                        ;(DATA_FLASH_START_ADDRESS) != #DEBUG_DATA

                MOV     DPTR,   #DATA_FLASH_START_ADDRESS
                MOV     A,      #DEBUG_DATA     ;Write DEBUG_DATA into Falsh
                ACALL   Byte_Program            ;Byte-program
                MOV     P1,     #11011111B      ;turn on P1.3 LED first, then turn on P1.5 LED
WAIT2:
                SJMP    WAIT2                   ;Stop here after byte-program
;*************************************************
;-------------------------------------------------------------------------
;Read one byte

Byte_Read:
                MOV     ISP_CONTR,      #ENABLE_ISP     ;Enable IAP/ISP function
                                                        ;determine Flash wait time
                MOV     ISP_CMD,        #01             ;Select Read AP mode
                MOV     ISP_ADDRH,      DPH         ;Fill page address in ISP_ADDRH&ISP_ADDRL
                MOV     ISP_ADDRL,      DPL
                CLR     EA
                MOV     ISP_TRIG,       #46H            ;Trigger ISP processing
                MOV     ISP_TRIG,       #0B9H           ;Trigger ISP processing
                NOP
                MOV     A,      ISP_DATA                ;Get the data in ISP_DATA
                SETB    EA
;Now in processing.(CPU will halt here before completing)
                ACALL   IAP_Disable                     ;Disable IAP function,
                                                        ;clear some registers associated with ISP
                RET
```

```
;-------------------------------------------------------------------------
;Byte-Program

Byte_Program:
                MOV     ISP_CONTR,      #ENABLE_ISP     ;Enable IAP function,
                                                        ;determine Flash wait time
                MOV     ISP_CMD,        #02H            ;Select Byte Program mode
                MOV     ISP_ADDRH,      DPH         ;Fill page address in ISP_ADDRH&ISP_ADDRL
                MOV     ISP_ADDRL,      DPL
                MOV     ISP_DATA,       A               ;Save the value into ISP_DATA
                CLR     EA
                MOV     ISP_TRIG,       #46H            ;Trigger ISP processing
                MOV     ISP_TRIG,       #0B9H           ;Trigger ISP processing
                NOP
                SETB    EA
                ACALL   IAP_Disable                     ;Disable IAP function,
                                                        ;clear some registers associated with ISP
                RET
;-------------------------------------------------------------------------
;Sector-Erase

Sector_Erase:
                MOV     ISP_CONTR,      #ENABLE_ISP     ;Enable IAP function,
                                                        ;determine Flash wait time
                MOV     ISP_CMD,        #03H            ;Select Page Erase Mode
                MOV     ISP_ADDRH,      DPH     ;Fill page address in ISP_ADDRH&ISP_ADDRL
                MOV     ISP_ADDRL,      DPL
                CLR     EA
                MOV     ISP_TRIG,       #46H            ;Trigger ISP processing
                MOV     ISP_TRIG,       #0B9H           ;Trigger ISP processing
                NOP
                SETB    EA
                ACALL   IAP_Disable                     ;Disable IAP function,
                                                        ;clear some registers associated with ISP
                RET
;-------------------------------------------------------------------------
Trigger_ISP:
                CLR     EA
                MOV     ISP_TRIG,       #46H            ;Trigger ISP processing
                MOV     ISP_TRIG,       #0B9H           ;Trigger ISP processing
                NOP
                SETB    EA
                RET
;-------------------------------------------------------------------------
IAP_Disable:                            ;Disable IAP function, cleal some registers associated with ISP
                MOV     ISP_CONTR,      #0
                MOV     ISP_CMD,        #0
                MOV     ISP_TRIG        #0
                RET
```

```
;-------------------------------------------------------------
Delay:
                CLR     A
                MOV     R0,     A
                MOV     R1,     A
                MOV     R2,     #20
Delay_loop:
                MOV     R0,     Delay_loop
                MOV     R1,     Delay_loop
                MOV     R2,     Delay_loop
                RET
;-------------------------------------------------------------
                END
;*****************************************
```

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.

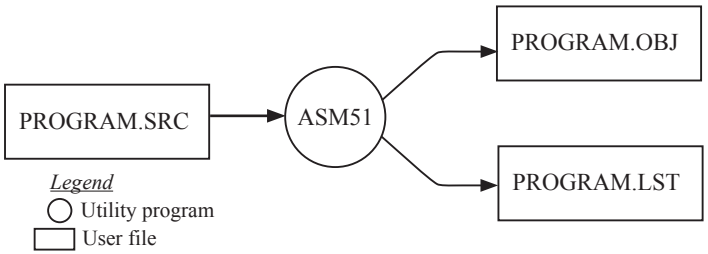


Figure 1 Assembling a source program

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

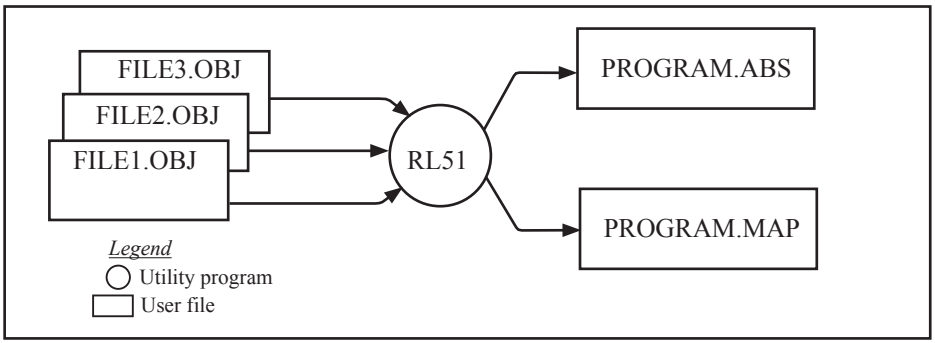
Assembler controls establish the format of the listing and object files by regulating the actions of ASM51. For the most part, assembler controls affect the look of the listing file, without having any affect on the program itself. They can be entered on the invocation line when a program is assembled,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

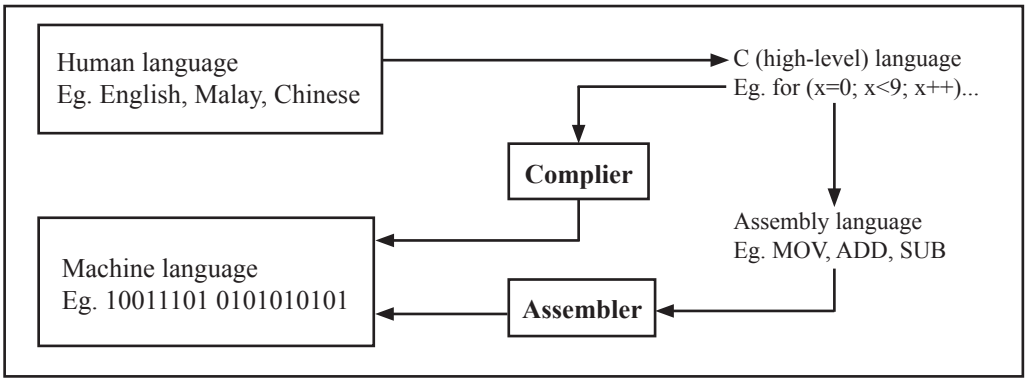
```
sbit portbit = P1^0;              /*Use variable portbit to refer to P1.0*/
main ( )
{
TMOD = 1;
while (1)
        {
        TH0 = 0xFE;
        TL0 = 0xC;
        TR0 = 1;
        while (TF0 !=1);
        TR0 = 0;
        TF0 = 0;
        portbit = !(P1.^0);
        }
}
```

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: STC89xx series Selection Table

Type 12T/6T 8051 MCU	Operating voltage (V)	Flash (B)	SARM (B)	TIMER	UART	DPTR	PCA/ PWM D/A	A/D	WDT	EEPROM (B)	Internal low voltage interrupt	Internal Reset threshold voltage can be configured	External interrupts which can wake up power down mode	Special timer for waking power down mode	Package of 40-pin (35 I/O ports)	Package of 44-pin (39 I/O ports)
STC89C51RC	5.5~3.3	4K	512	3	1	2	N	N	Y	4K	Y	N	4	N	PDIP	LQFP/ PLCC
STC89C52RC	5.5~3.3	8K	512	3	1	2	N	N	Y	4K	Y	N	4	N	PDIP	LQFP/ PLCC
STC89C53RC	5.5~3.3	13K	512	3	1	2	N	N	Y	/	Y	N	4	N	PDIP	LQFP/ PLCC
STC89LE51RC	3.6~2.0	4K	512	3	1	2	N	N	Y	4K	Y	N	4	N	PDIP	LQFP/ PLCC
STC89LE52RC	3.6~2.0	8K	512	3	1	2	N	N	Y	4K	Y	N	4	N	PDIP	LQFP/ PLCC
STC89LE53RC	3.6~2.0	13K	512	3	1	2	N	N	Y	/	Y	N	4	N	PDIP	LQFP/ PLCC
STC89C54RD+	5.5~3.3	16K	1280	3	1	2	N	N	Y	45K	Y	N	4	N	PDIP	LQFP/ PLCC
STC89C58RD+	5.5~3.3	32K	1280	3	1	2	N	N	Y	29K	Y	N	4	N	PDIP	LQFP/ PLCC
STC89C516RD+	5.5~3.3	61K	1280	3	1	2	N	N	Y	/	Y	N	4	N	PDIP	LQFP/ PLCC
STC89LE54RD+	3.6~2.0	16K	1280	3	1	2	N	N	Y	45K	Y	N	4	N	PDIP	LQFP/ PLCC
STC89LE58RD+	3.6~2.0	32K	1280	3	1	2	N	N	Y	29K	Y	N	4	N	PDIP	LQFP/ PLCC
STC89LE516RD+	3.6~2.0	61K	1280	3	1	2	N	N	Y	/	Y	N	4	N	PDIP	LQFP/ PLCC