========== ==延时的 Delay_ms.asm===========

```
stm8/
    #include "mapping.inc"
    #include "stm8s105s6.inc"
      #include "D:\STM8 程序 我的 STM8 程序 \GPIO\User_register.inc"
      #include "D:\STM8 程序 我的 STM8 程序 \GPIO\define.inc"
      segment'rom'
;*************       延时程序 **************
;CPU 频率:2MHz
;要用到的寄存器 :X,Y
;把要延时的值写入   R0E.R0F,R0E为高位值 .R0F 为低位值
;1ms

Delay_ms.L
      PUSHW X
      PUSHW Y
      PUSH CC
   LDW X,#500                ;延时 1S
      LDW Y,R0E
Delay_next1:
   LDW X,#500
Delay_next:
   DECW X
   JRNE Delay_next
   DECW Y
   JRNE Delay_next1
     POP CC
     POPW Y
     POPW X
     RETF
     RETF
     RETF

end
```

========== =mai.asm==========

```
stm8/

      #include "mapping.inc"
   #include "stm8s105s6.inc"
      #include "D:\STM8 程序 我的 STM8 程序 \GPIO\User_register.inc"
      #include "D:\STM8 程序 我的 STM8 程序 \GPIO\define.inc"
```

```
        segment'rom'


main.l
        ; initialize  SP
        ldw X,#stack_end
        ldw SP,X

        #ifdef RAM0
        ; clear RAM0
ram0_start.b EQU $ram0_segment_start
ram0_end.b EQU $ram0_segment_end
        ldw X,#ram0_start
clear_ram0.l
        clr (X)
        incw X
        cpw X,#ram0_end
        jrule clear_ram0
        #endif

        #ifdef RAM1
        ; clear RAM1
ram1_start.w EQU $ram1_segment_start
ram1_end.w EQU $ram1_segment_end
        ldw X,#ram1_start
clear_ram1.l
        clr (X)
        incw X
        cpw X,#ram1_end
        jrule clear_ram1
        #endif

        ; clear stack
stack_start.w EQU $stack_segment_start
stack_end.w EQU $stack_segment_end
        ldw X,#stack_start
clear_stack.l
        clr (X)
        incw X
        cpw X,#stack_end
        jrule clear_stack
    intel                               ;主函数开始
;  更换成外部时钟  ,并且  CPU8 分频
        BSET CLK_SWCR,#1            ;允许更换时钟
```

```
    BRES CLK_SWCR,#2          ;SWIEN 位为 0,用查询方式确定时钟更换是否完
成
    MOV  CLK_SWR,#0B4H        ;目标时钟为  HSE 晶振
CLK_SW_WAIT1:                          ;等待时钟更换中断标志   SWIF 有效
   BTJF CLK_SWCR,#3,CLK_SW_WAIT1
    BRES CLK_SWCR,#3              ;清除时钟更换完成任务中断标志     SWIF
    BRES CLK_SWCR,#1              ;SWEN 位为 0,禁止时钟再次更换
    BRES CLK_ICKR,#0          ;关闭 HSI 时钟 ,减少功耗

    MOV  CLK_CKDIVR,#1bH      ;内部时钟  8 分频,CPU 时钟  128 分频,现在使用
外部时钟 ,前 8 分频没用
    BSET CLK_CSSR,#0          ;CSS时钟安全系统开

;I/O  口初始化
   BSET PD_DDR_DDR0           ;PD_0 设置成输出
    BSET PE_DDR_DDR5          ;PE_5 设置成输出 ,595 的 SDATA 脚
    BSET PC_DDR_DDR2          ;PC_2 设置成输出 ,595 的 SCLK 脚
    BSET PC_DDR_DDR4          ;PC_4 设置成输出 ,595 的 SRCK 脚
    BSET PE_CR1_C5            ;设置成推换
    BSET PC_CR1_C2            ;设置成推换
    BSET PC_CR1_C4            ;设置成推换
    BSET PD_CR1_C0            ;PD_0 设置成推换
    BSET PE_CR2_C5            ;PE_5 设置成高速
    BSET PC_CR2_C2            ;PC_2 设置成高速
    BSET PC_CR2_C4            ;PC_4 设置成高速
    BSET PD_CR2_C0                ;PD_0 设置成高速
;数码管要显示的数字
    MOV  R0D,#9               ;R0A 为高位
    MOV  R0C,#8
    MOV  R0B,#6
    MOV  R0A,#7

    CALLF  Display_595        ;调用 595 显示


LOOP1:
MOV  R0E,#02H       ;R0E 和  R0F 两单元合起来对应十进制的    2000
MOV  R0F,#0FFH
   BSET PD_ODR_ODR0
   CALLF  Delay_ms            ;调用延时
   BRES PD_ODR_ODR0
CALLF  Delay_ms            ;调用延时
   CALLF  AD_8BIT              ;调用  AD  转换
   CALLF  BCD_8bit             ;调用数据转换
```

```
        CALLF  Display_595          ;调用 595 显示
        JPLOOP1

;*******************8          位  二  进  制  的  数  值  转  化  为  BCD  码
**************************
;把 8 位二进制的数值转化为   BCD 码
;要转换的数值放在   R09 中
;转换结果的高位在   R0A,中位在  R0B,低位在  R0C
BCD_8bit.L
        PUSH A
        PUSH R01
        LD  A,R09
        LD  XL,A
        LD  A,#100
        DIV  X,A
        LD  R01,A
        LD  A,XL
        LD  R0A,A
        LD  A,R01
        LD  XL,A
        LD  A,#10
        DIV  X,A
        LD R0C,A
        LD A,XL
        LD R0B,A
        POPR01
        POPA
        RETF
        RETF
        RETF
·*****************************************************************
'
*******

;*************        延时程序 **************
;CPU 频率:2MHz
;要用到的寄存器  :X,Y
;把要延时的值写入   R0E.R0F,R0E为高位值 .R0F 为低位值
;1ms

Delay_ms.L
        PUSHW X
```

```
        PUSHW Y
        PUSH CC
    LDW  X,#500              ;延时 1S
        LDW  Y,R0E
Delay_next1:
    LDW  X,#500
Delay_next:
    DECW  X
    JRNE Delay_next
    DECW  Y
    JRNE Delay_next1
        POPCC
        POPW Y
        POPW X
        RETF
        RETF
        RETF
;*****************************************

;595 串行数码管显示  _4 位
;四位的值分别装在    R0A,R0B,R0C,R0D 单元中 ,高位在前
;要用到  A,X  寄存?
;过程用到的寄存器  :R0F

Display_595.L
    PUSH A
        PUSH CC
        PUSH R0F
        PUSHW X
        CLRW  X
        LD  A,R0D
        LD  XL,A
    LD  A,(TABLE_LED,X)
        MOV  R0F,#8
Display_loop1:
    BSET PC_ODR_ODR2
    RLC A
    BCCM  PE_ODR_ODR5
    NOP
    BRES PC_ODR_ODR2
    DEC  R0F
    JRNE Display_loop1
    ;送完一个数据
    CLRW  X
```

```
        LD  A,R0C
        LD  XL,A
        LD  A,(TABLE_LED,X)
        MOV  R0F,#8
Display_loop2:
    BSET PC_ODR_ODR2
        RLC A
        BCCM  PE_ODR_ODR5
        NOP
        BRES PC_ODR_ODR2
        DEC  R0F
        JRNE Display_loop2

        CLRW  X
        LD  A,R0B
        LD  XL,A
        LD  A,(TABLE_LED,X)
        MOV  R0F,#8
Display_loop3:
    BSET PC_ODR_ODR2
        RLC A
        BCCM  PE_ODR_ODR5
        NOP
        BRES PC_ODR_ODR2
        DEC  R0F
        JRNE Display_loop3


        CLRW  X
        LD  A,R0A
        LD  XL,A
        LD  A,(TABLE_LED,X)
        MOV  R0F,#8
Display_loop4:
    BSET PC_ODR_ODR2
        RLC A
        BCCM  PE_ODR_ODR5    ;SDATA
        NOP
        BRES PC_ODR_ODR2    ;SCLK
        DEC  R0F
        JRNE Display_loop4
        ;4 个数据送完成
        BSET PC_ODR_ODR2
        BRES PC_ODR_ODR4     ;置高  SRCK
```

```
NOP
BSET PC_ODR_ODR4      ;输出数据

POPW X
POP R0F
POP CC
POP A
RETF
RETF
      RETF


;*************************************************************
;AD  转换
;入口参数 :对 PB_7 进行转换
;出口参数 :转换结果放大   R09
AD_8BIT.L
   PUSH A
      PUSH CC
      PUSHW X
      ;引脚初始化
      BRES PB_DDR_DDR7        ;输入
      BRES PB_CR1_C7          ;浮空
      BRES PB_CR2_C7          ;禁止外中断
   ;配置寄存器
   MOV  ADC_CSR,#07H       ;禁止转换结束中断   ,通道为  AIN7;
   MOV  ADC_CR2,#00H       ;禁止外部触发 ,数据高  8 位对好
   MOV  ADC_CR3,#00H       ;禁止数据缓存使能
      MOV  ADC_CR1,#71H       ;转换速度为最慢   ,单次转换模式  ,开启  AD  转换电
源
      MOV  ADC_CR1,#71H       ;开始转换
      NOP
      NOP
      NOP
AD_next:
   LD  A,ADC_CSR
  AND  A,#80H
    JREQ AD_next
    MOV  ADC_CSR,#07H       ;清除转换结束标志
    MOV  ADC_CR1,#70H       ;停止  AD  转换
    MOV  R09,ADC_DRH         ;把数据送到   R09 单元
    POPW X
    POP CC
    POP A
```

```
        RETF
        RETF
        RETF
;*****************************************************************
;
;*****************************************************************
;595 的显示数表
TABLE_LED:      ;DP,F,G,E,D,C,B,A
          ;   0 ,   1   ,   2   , 3 , 4 , 5 , 6 , 7 ,   8 , 9
    DC.B 0A0H,0F9H,0C4H,0D0H,99H,92H,82H,0F8H,80H,90H
;*****************************************************************


    motorola
        interrupt NonHandledInterrupt
NonHandledInterrupt.l
        iret

        segment'vectit'
        dc.l {$82000000+main}                                   ; reset
        dc.l {$82000000+NonHandledInterrupt}    ; trap
        dc.l {$82000000+NonHandledInterrupt}    ; irq0
        dc.l {$82000000+NonHandledInterrupt}    ; irq1
        dc.l {$82000000+NonHandledInterrupt}    ; irq2
        dc.l {$82000000+NonHandledInterrupt}    ; irq3
        dc.l {$82000000+NonHandledInterrupt}    ; irq4
        dc.l {$82000000+NonHandledInterrupt}    ; irq5
        dc.l {$82000000+NonHandledInterrupt}    ; irq6
        dc.l {$82000000+NonHandledInterrupt}    ; irq7
        dc.l {$82000000+NonHandledInterrupt}    ; irq8
        dc.l {$82000000+NonHandledInterrupt}    ; irq9
        dc.l {$82000000+NonHandledInterrupt}    ; irq10
        dc.l {$82000000+NonHandledInterrupt}    ; irq11
        dc.l {$82000000+NonHandledInterrupt}    ; irq12
        dc.l {$82000000+NonHandledInterrupt}    ; irq13
        dc.l {$82000000+NonHandledInterrupt}    ; irq14
        dc.l {$82000000+NonHandledInterrupt}    ; irq15
        dc.l {$82000000+NonHandledInterrupt}    ; irq16
        dc.l {$82000000+NonHandledInterrupt}    ; irq17
        dc.l {$82000000+NonHandledInterrupt}    ; irq18
        dc.l {$82000000+NonHandledInterrupt}    ; irq19
        dc.l {$82000000+NonHandledInterrupt}    ; irq20
```

```
dc.l {$82000000+NonHandledInterrupt}      ; irq21
dc.l {$82000000+NonHandledInterrupt}      ; irq22
dc.l {$82000000+NonHandledInterrupt}      ; irq23
dc.l {$82000000+NonHandledInterrupt}      ; irq24
dc.l {$82000000+NonHandledInterrupt}      ; irq25
dc.l {$82000000+NonHandledInterrupt}      ; irq26
dc.l {$82000000+NonHandledInterrupt}      ; irq27
dc.l {$82000000+NonHandledInterrupt}      ; irq28
dc.l {$82000000+NonHandledInterrupt}      ; irq29

        end
==========================User_register.asm=======================
=
stm8/


;用户定义公用变量
        segment'ram0'
        BYTES
;00-0F 单元定义为字节变量  (供主程序使用 )
.R00.B ds.b 1
.R01.B ds.b 1
.R02.B ds.b 1
.R03.B ds.b 1
.R04.B ds.b 1
.R05.B ds.b 1
.R06.B ds.b 1
.R07.B ds.b 1
.R08.B ds.b 1
.R09.B ds.b 1
.R0A.B  ds.b 1
.R0B.B  ds.b 1
.R0C.B ds.b 1
.R0D.B  ds.b 1
.R0E.B ds.b 1
.R0F.B ds.b 1

;10-1F 单元定义为字节变量  (供优先级为  1 中断服务程序使用  )
.R10.B ds.b 1
.R11.B ds.b 1
.R12.B ds.b 1
.R13.B ds.b 1
.R14.B ds.b 1
.R15.B ds.b 1
```

```
.R16.B ds.b 1
.R17.B ds.b 1
.R18.B ds.b 1
.R19.B ds.b 1
.R1A.B  ds.b 1
.R1B.B  ds.b 1
.R1C.B ds.b 1
.R1D.B  ds.b 1
.R1E.B ds.b 1
.R1F.B ds.b 1

;20-2F 单元定义为字节变量  (供优先级为  2 中断服务程序使用  )
.R20.B ds.b 1
.R21.B ds.b 1
.R22.B ds.b 1
.R23.B ds.b 1
.R24.B ds.b 1
.R25.B ds.b 1
.R26.B ds.b 1
.R27.B ds.b 1
.R28.B ds.b 1
.R29.B ds.b 1
.R2A.B  ds.b 1
.R2B.B  ds.b 1
.R2C.B ds.b 1
.R2D.B  ds.b 1
.R2E.B  ds.b 1
.R2F.B ds.b 1

;30-3F 单元定义为字节变量  (供优先级为  3 中断服务程序使用  )
.R30.B ds.b 1
.R31.B ds.b 1
.R32.B ds.b 1
.R33.B ds.b 1
.R34.B ds.b 1
.R35.B ds.b 1
.R36.B ds.b 1
.R37.B ds.b 1
.R38.B ds.b 1
.R39.B ds.b 1
.R3A.B  ds.b 1
.R3B.B  ds.b 1
.R3C.B ds.b 1
.R3D.B  ds.b 1
```

```
.R3E.B ds.b 1
.R3F.B ds.b 1
;用户定义公用变量
        segment'ram1'
        WORDS
;00-0F 单元定义为字节变量  (供主程序使用 )
.R40.W ds.w 1
.R41.W ds.w 1
.R42.W ds.w 1
.R43.W ds.w 1
.R44.W ds.w 1
.R45.W ds.w 1
.R46.W ds.w 1
.R47.W ds.w 1
.R48.W ds.w 1
.R49.W ds.w 1
.R4A.W  ds.w 1
.R4B.W ds.w 1
.R4C.W ds.w 1
.R4D.W  ds.w 1
.R4E.W ds.w 1
.R4F.W ds.w 1
        end
=========================define.inc      I/O 口引脚 ================

;GPIO 部分寄存器宏定义
;*****************************************************************
**********************************
;    GPIO          PX_ODR
;*****************************************************************
**********************************
;PA 口输出数据锁存
  #define PA_ODR_ODR0 PA_ODR,#0    ;PA 口 0 位输出数据
  #define PA_ODR_ODR1 PA_ODR,#1    ;PA 口 1 位输出数据
  #define PA_ODR_ODR2 PA_ODR,#2    ;PA 口 2 位输出数据
  #define PA_ODR_ODR3 PA_ODR,#3    ;PA 口 3 位输出数据
  #define PA_ODR_ODR4 PA_ODR,#4    ;PA 口 4 位输出数据
  #define PA_ODR_ODR5 PA_ODR,#5    ;PA 口 5 位输出数据
  #define PA_ODR_ODR6 PA_ODR,#6    ;PA 口 7 位输出数据
  #define PA_ODR_ODR7 PA_ODR,#7    ;PA 口 7 位输出数据

;PB 口输出数据锁存
  #define PB_ODR_ODR0 PB_ODR,#0    ;PB 口 0 位输出数据
  #define PB_ODR_ODR1 PB_ODR,#1    ;PB 口 1 位输出数据
```

```
  #define PB_ODR_ODR2 PB_ODR,#2    ;PB 口 2 位输出数据
  #define PB_ODR_ODR3 PB_ODR,#3    ;PB 口 3 位输出数据
  #define PB_ODR_ODR4 PB_ODR,#4    ;PB 口 4 位输出数据
  #define PB_ODR_ODR5 PB_ODR,#5    ;PB 口 5 位输出数据
  #define PB_ODR_ODR6 PB_ODR,#6    ;PB 口 7 位输出数据
  #define PB_ODR_ODR7 PB_ODR,#7    ;PB 口 7 位输出数据

;PC 口输出数据锁存
  #define PC_ODR_ODR0 PC_ODR,#0    ;PC 口 0 位输出数据
  #define PC_ODR_ODR1 PC_ODR,#1    ;PC 口 1 位输出数据
  #define PC_ODR_ODR2 PC_ODR,#2    ;PC 口 2 位输出数据
  #define PC_ODR_ODR3 PC_ODR,#3    ;PC 口 3 位输出数据
  #define PC_ODR_ODR4 PC_ODR,#4    ;PC 口 4 位输出数据
  #define PC_ODR_ODR5 PC_ODR,#5    ;PC 口 5 位输出数据
  #define PC_ODR_ODR6 PC_ODR,#6    ;PC 口 7 位输出数据
  #define PC_ODR_ODR7 PC_ODR,#7    ;PC 口 7 位输出数据

;PD 口输出数据锁存
  #define PD_ODR_ODR0 PD_ODR,#0    ;PD 口 0 位输出数据
  #define PD_ODR_ODR1 PD_ODR,#1    ;PD 口 1 位输出数据
  #define PD_ODR_ODR2 PD_ODR,#2    ;PD 口 2 位输出数据
  #define PD_ODR_ODR3 PD_ODR,#3    ;PD 口 3 位输出数据
  #define PD_ODR_ODR4 PD_ODR,#4    ;PD 口 4 位输出数据
  #define PD_ODR_ODR5 PD_ODR,#5    ;PD 口 5 位输出数据
  #define PD_ODR_ODR6 PD_ODR,#6    ;PD 口 7 位输出数据
  #define PD_ODR_ODR7 PD_ODR,#7    ;PD 口 7 位输出数据

;PE 口输出数据锁存
  #define PE_ODR_ODR0 PE_ODR,#0    ;PE 口 0 位输出数据
  #define PE_ODR_ODR1 PE_ODR,#1    ;PE 口 1 位输出数据
  #define PE_ODR_ODR2 PE_ODR,#2    ;PE 口 2 位输出数据
  #define PE_ODR_ODR3 PE_ODR,#3    ;PE 口 3 位输出数据
  #define PE_ODR_ODR4 PE_ODR,#4    ;PE 口 4 位输出数据
  #define PE_ODR_ODR5 PE_ODR,#5    ;PE 口 5 位输出数据
  #define PE_ODR_ODR6 PE_ODR,#6    ;PE 口 7 位输出数据
  #define PE_ODR_ODR7 PE_ODR,#7    ;PE 口 7 位输出数据

;*****************************************************************
**********************************
;    GPIO            PX_IDR
;*****************************************************************
**********************************
;PA 口输入数据寄存器  ,用来读取  I/O 口的电平状态
```

```
#define PA_IDR_IDR0  PA_IDR,#0    ;PA 口对应位输入数据
#define PA_IDR_IDR1  PA_IDR,#1    ;PA 口对应位输入数据
#define PA_IDR_IDR2  PA_IDR,#2    ;PA 口对应位输入数据
#define PA_IDR_IDR3  PA_IDR,#3    ;PA 口对应位输入数据
#define PA_IDR_IDR4  PA_IDR,#4    ;PA 口对应位输入数据
#define PA_IDR_IDR5  PA_IDR,#5    ;PA 口对应位输入数据
#define PA_IDR_IDR6  PA_IDR,#6    ;PA 口对应位输入数据
#define PA_IDR_IDR7  PA_IDR,#7    ;PA 口对应位输入数据

;PB 口输入数据寄存器 ,用来读取 I/O 口的电平状态
  #define PB_IDR_IDR0  PB_IDR,#0    ;PB 口对应位输入数据
  #define PB_IDR_IDR1  PB_IDR,#1    ;PB 口对应位输入数据
  #define PB_IDR_IDR2  PB_IDR,#2    ;PB 口对应位输入数据
  #define PB_IDR_IDR3  PB_IDR,#3    ;PB 口对应位输入数据
  #define PB_IDR_IDR4  PB_IDR,#4    ;PB 口对应位输入数据
  #define PB_IDR_IDR5  PB_IDR,#5    ;PB 口对应位输入数据
  #define PB_IDR_IDR6  PB_IDR,#6    ;PB 口对应位输入数据
  #define PB_IDR_IDR7  PB_IDR,#7    ;PB 口对应位输入数据

;PC 口输入数据寄存器 ,用来读取 I/O 口的电平状态
  #define PC_IDR_IDR0  PC_IDR,#0    ;PC 口对应位输入数据
  #define PC_IDR_IDR1  PC_IDR,#1    ;PC 口对应位输入数据
  #define PC_IDR_IDR2  PC_IDR,#2    ;PC 口对应位输入数据
  #define PC_IDR_IDR3  PC_IDR,#3    ;PC 口对应位输入数据
  #define PC_IDR_IDR4  PC_IDR,#4    ;PC 口对应位输入数据
  #define PC_IDR_IDR5  PC_IDR,#5    ;PC 口对应位输入数据
  #define PC_IDR_IDR6  PC_IDR,#6    ;PC 口对应位输入数据
  #define PC_IDR_IDR7  PC_IDR,#7    ;PC 口对应位输入数据

;PD 口输入数据寄存器 ,用来读取 I/O 口的电平状态
  #define PD_IDR_IDR0  PD_IDR,#0    ;PD 口对应位输入数据
  #define PD_IDR_IDR1  PD_IDR,#1    ;PD 口对应位输入数据
  #define PD_IDR_IDR2  PD_IDR,#2    ;PD 口对应位输入数据
  #define PD_IDR_IDR3  PD_IDR,#3    ;PD 口对应位输入数据
  #define PD_IDR_IDR4  PD_IDR,#4    ;PD 口对应位输入数据
  #define PD_IDR_IDR5  PD_IDR,#5    ;PD 口对应位输入数据
  #define PD_IDR_IDR6  PD_IDR,#6    ;PD 口对应位输入数据
  #define PD_IDR_IDR7  PD_IDR,#7    ;PD 口对应位输入数据

;PE 口输入数据寄存器 ,用来读取 I/O 口的电平状态
  #define PE_IDR_IDR0  PE_IDR,#0    ;PE 口对应位输入数据
  #define PE_IDR_IDR1  PE_IDR,#1    ;PE 口对应位输入数据
  #define PE_IDR_IDR2  PE_IDR,#2    ;PE 口对应位输入数据
  #define PE_IDR_IDR3  PE_IDR,#3    ;PE 口对应位输入数据
```

```
  #define PE_IDR_IDR4  PE_IDR,#4    ;PE 口对应位输入数据
  #define PE_IDR_IDR5  PE_IDR,#5    ;PE 口对应位输入数据
  #define PE_IDR_IDR6  PE_IDR,#6    ;PE 口对应位输入数据
  #define PE_IDR_IDR7  PE_IDR,#7    ;PE 口对应位输入数据


;*****************************************************************
**********************************
;    GPIO           PX_DDR
;*****************************************************************
**********************************
;PA 数据传输方向控制寄存器
  #define PA_DDR_DDR0  PA_DDR,#0    ;PA 口对应位数据传输方向控制    __0:输入
1:输出
  #define PA_DDR_DDR1  PA_DDR,#1    ;PA 口对应位数据传输方向控制    __0:输入
1:输出
  #define PA_DDR_DDR2  PA_DDR,#2    ;PA 口对应位数据传输方向控制    __0:输入
1:输出
  #define PA_DDR_DDR3  PA_DDR,#3    ;PA 口对应位数据传输方向控制    __0:输入
1:输出
  #define PA_DDR_DDR4  PA_DDR,#4    ;PA 口对应位数据传输方向控制    __0:输入
1:输出
  #define PA_DDR_DDR5  PA_DDR,#5    ;PA 口对应位数据传输方向控制    __0:输入
1:输出
  #define PA_DDR_DDR6  PA_DDR,#6    ;PA 口对应位数据传输方向控制    __0:输入
1:输出
  #define PA_DDR_DDR7  PA_DDR,#7    ;PA 口对应位数据传输方向控制    __0:输入
1:输出

;PB 数据传输方向控制寄存器
  #define PB_DDR_DDR0  PB_DDR,#0    ;PB 口对应位数据传输方向控制    __0:输入
1:输出
  #define PB_DDR_DDR1  PB_DDR,#1    ;PB 口对应位数据传输方向控制    __0:输入
1:输出
  #define PB_DDR_DDR2  PB_DDR,#2    ;PB 口对应位数据传输方向控制    __0:输入
1:输出
  #define PB_DDR_DDR3  PB_DDR,#3    ;PB 口对应位数据传输方向控制    __0:输入
1:输出
  #define PB_DDR_DDR4  PB_DDR,#4    ;PB 口对应位数据传输方向控制    __0:输入
1:输出
  #define PB_DDR_DDR5  PB_DDR,#5    ;PB 口对应位数据传输方向控制    __0:输入
1:输出
  #define PB_DDR_DDR6  PB_DDR,#6    ;PB 口对应位数据传输方向控制    __0:输入
1:输出
```

```
  #define PB_DDR_DDR7  PB_DDR,#7    ;PB 口对应位数据传输方向控制    __0:输入
1:输出

;PC 数据传输方向控制寄存器
  #define PC_DDR_DDR0  PC_DDR,#0    ;PC 口对应位数据传输方向控制    __0:输入
1:输出
  #define PC_DDR_DDR1  PC_DDR,#1    ;PC 口对应位数据传输方向控制    __0:输入
1:输出
  #define PC_DDR_DDR2  PC_DDR,#2    ;PC 口对应位数据传输方向控制    __0:输入
1:输出
  #define PC_DDR_DDR3  PC_DDR,#3    ;PC 口对应位数据传输方向控制    __0:输入
1:输出
  #define PC_DDR_DDR4  PC_DDR,#4    ;PC 口对应位数据传输方向控制    __0:输入
1:输出
  #define PC_DDR_DDR5  PC_DDR,#5    ;PC 口对应位数据传输方向控制    __0:输入
1:输出
  #define PC_DDR_DDR6  PC_DDR,#6    ;PC 口对应位数据传输方向控制    __0:输入
1:输出
  #define PC_DDR_DDR7  PC_DDR,#7    ;PC 口对应位数据传输方向控制    __0:输入
1:输出

;PD 数据传输方向控制寄存器
  #define PD_DDR_DDR0  PD_DDR,#0    ;PD 口对应位数据传输方向控制    __0:输入
1:输出
  #define PD_DDR_DDR1  PD_DDR,#1    ;PD 口对应位数据传输方向控制    __0:输入
1:输出
  #define PD_DDR_DDR2  PD_DDR,#2    ;PD 口对应位数据传输方向控制    __0:输入
1:输出
  #define PD_DDR_DDR3  PD_DDR,#3    ;PD 口对应位数据传输方向控制    __0:输入
1:输出
  #define PD_DDR_DDR4  PD_DDR,#4    ;PD 口对应位数据传输方向控制    __0:输入
1:输出
  #define PD_DDR_DDR5  PD_DDR,#5    ;PD 口对应位数据传输方向控制    __0:输入
1:输出
  #define PD_DDR_DDR6  PD_DDR,#6    ;PD 口对应位数据传输方向控制    __0:输入
1:输出
  #define PD_DDR_DDR7  PD_DDR,#7    ;PD 口对应位数据传输方向控制    __0:输入
1:输出

;PE 数据传输方向控制寄存器
  #define PE_DDR_DDR0  PE_DDR,#0    ;PE 口对应位数据传输方向控制    __0:输入
1:输出
  #define PE_DDR_DDR1  PE_DDR,#1    ;PE 口对应位数据传输方向控制    __0:输入
1:输出
```

```
  #define PE_DDR_DDR2  PE_DDR,#2    ;PE 口对应位数据传输方向控制   __0:输入
1:输出
  #define PE_DDR_DDR3  PE_DDR,#3    ;PE 口对应位数据传输方向控制   __0:输入
1:输出
  #define PE_DDR_DDR4  PE_DDR,#4    ;PE 口对应位数据传输方向控制   __0:输入
1:输出
  #define PE_DDR_DDR5  PE_DDR,#5    ;PE 口对应位数据传输方向控制   __0:输入
1:输出
  #define PE_DDR_DDR6  PE_DDR,#6    ;PE 口对应位数据传输方向控制   __0:输入
1:输出
  #define PE_DDR_DDR7  PE_DDR,#7    ;PE 口对应位数据传输方向控制   __0:输入
1:输出


;*****************************************************************************
**********************************
;    GPIO           PX_CR1
;*****************************************************************************
**********************************
;PA 相应端口模式控制寄存器
;PA 口对应位模式控制寄存器   :当 DDR=0__输入时，   0:浮空输入 ,1:上拉输入 ;
DDR=1__输出时 ,0:开漏输出 ,1:推挽输出
  #define PA_CR1_C0     PA_CR1,#0
  #define PA_CR1_C1     PA_CR1,#1
  #define PA_CR1_C2     PA_CR1,#2
  #define PA_CR1_C3     PA_CR1,#3
  #define PA_CR1_C4     PA_CR1,#4
  #define PA_CR1_C5     PA_CR1,#5
  #define PA_CR1_C6     PA_CR1,#6
  #define PA_CR1_C7     PA_CR1,#7

;PB 相应端口模式控制寄存器
;PB 口对应位模式控制寄存器   :当 DDR=0__输入时，   0:浮空输入 ,1:上拉输入 ;
DDR=1__输出时 ,0:开漏输出 ,1:推挽输出
  #define PB_CR1_C0     PB_CR1,#0
  #define PB_CR1_C1     PB_CR1,#1
  #define PB_CR1_C2     PB_CR1,#2
  #define PB_CR1_C3     PB_CR1,#3
  #define PB_CR1_C4     PB_CR1,#4
  #define PB_CR1_C5     PB_CR1,#5
  #define PB_CR1_C6     PB_CR1,#6
  #define PB_CR1_C7     PB_CR1,#7

;PC 相应端口模式控制寄存器
```

```
;PC 口对应位模式控制寄存器  :当 DDR=0__输入时 ,   0:浮空输入 ,1:上拉输入 ;
DDR=1__ 输出时 ,0:开漏输出 ,1:推挽输出
  #define PC_CR1_C0     PC_CR1,#0
  #define PC_CR1_C1     PC_CR1,#1
  #define PC_CR1_C2     PC_CR1,#2
  #define PC_CR1_C3     PC_CR1,#3
  #define PC_CR1_C4     PC_CR1,#4
  #define PC_CR1_C5     PC_CR1,#5
  #define PC_CR1_C6     PC_CR1,#6
  #define PC_CR1_C7     PC_CR1,#7

;PD 相应端口模式控制寄存器
;PD 口对应位模式控制寄存器  :当 DDR=0__输入时 ,   0:浮空输入 ,1:上拉输入 ;
DDR=1__ 输出时 ,0:开漏输出 ,1:推挽输出
  #define PD_CR1_C0     PD_CR1,#0
  #define PD_CR1_C1     PD_CR1,#1
  #define PD_CR1_C2     PD_CR1,#2
  #define PD_CR1_C3     PD_CR1,#3
  #define PD_CR1_C4     PD_CR1,#4
  #define PD_CR1_C5     PD_CR1,#5
  #define PD_CR1_C6     PD_CR1,#6
  #define PD_CR1_C7     PD_CR1,#7

;PE 相应端口模式控制寄存器
;PE 口对应位模式控制寄存器  :当 DDR=0__ 输入时 ,   0:浮空输入 ,1:上拉输入 ;
DDR=1__ 输出时 ,0:开漏输出 ,1:推挽输出
  #define PE_CR1_C0     PE_CR1,#0
  #define PE_CR1_C1     PE_CR1,#1
  #define PE_CR1_C2     PE_CR1,#2
  #define PE_CR1_C3     PE_CR1,#3
  #define PE_CR1_C4     PE_CR1,#4
  #define PE_CR1_C5     PE_CR1,#5
  #define PE_CR1_C6     PE_CR1,#6
  #define PE_CR1_C7     PE_CR1,#7

;*****************************************************************************
**********************************
;    GPIO          PX_CR2
;*****************************************************************************
**********************************
;PA 品相应位输入中断或输出速度控制寄存器
;说明:当 DDR=0__ 输入时   0:禁止外部中断    1:全能外部中断    当 DDR=1__输出时
0:最大速度  2MHZ  1:最大速度  10MHZ
```

```
#define PA_CR2_C0    PA_CR2,#0
#define PA_CR2_C1    PA_CR2,#1
#define PA_CR2_C2    PA_CR2,#2
#define PA_CR2_C3    PA_CR2,#3
#define PA_CR2_C4    PA_CR2,#4
#define PA_CR2_C5    PA_CR2,#5
#define PA_CR2_C6    PA_CR2,#6
#define PA_CR2_C7    PA_CR2,#7

;PB 品相应位输入中断或输出速度控制寄存器
;说明:当 DDR=0__输入时  0:禁止外部中断   1:全能外部中断   当 DDR=1__输出时
0:最大速度  2MHZ  1:最大速度  10MHZ
  #define PB_CR2_C0    PB_CR2,#0
  #define PB_CR2_C1    PB_CR2,#1
  #define PB_CR2_C2    PB_CR2,#2
  #define PB_CR2_C3    PB_CR2,#3
  #define PB_CR2_C4    PB_CR2,#4
  #define PB_CR2_C5    PB_CR2,#5
  #define PB_CR2_C6    PB_CR2,#6
  #define PB_CR2_C7    PB_CR2,#7

;PC 品相应位输入中断或输出速度控制寄存器
;说明:当 DDR=0__输入时  0:禁止外部中断   1:全能外部中断   当 DDR=1__输出时
0:最大速度  2MHZ  1:最大速度  10MHZ
  #define PC_CR2_C0    PC_CR2,#0
  #define PC_CR2_C1    PC_CR2,#1
  #define PC_CR2_C2    PC_CR2,#2
  #define PC_CR2_C3    PC_CR2,#3
  #define PC_CR2_C4    PC_CR2,#4
  #define PC_CR2_C5    PC_CR2,#5
  #define PC_CR2_C6    PC_CR2,#6
  #define PC_CR2_C7    PC_CR2,#7

;PD 品相应位输入中断或输出速度控制寄存器
;说明:当 DDR=0__输入时  0:禁止外部中断   1:全能外部中断   当 DDR=1__输出时
0:最大速度  2MHZ  1:最大速度  10MHZ
#define PD_CR2_C0    PD_CR2,#0
#define PD_CR2_C1    PD_CR2,#1
#define PD_CR2_C2    PD_CR2,#2
#define PD_CR2_C3    PD_CR2,#3
#define PD_CR2_C4    PD_CR2,#4
#define PD_CR2_C5    PD_CR2,#5
#define PD_CR2_C6    PD_CR2,#6
#define PD_CR2_C7    PD_CR2,#7
```

```
;PE 品相应位输入中断或输出速度控制寄存器
;说明:当 DDR=0__输入时   0:禁止外部中断    1:全能外部中断    当 DDR=1__输出时
0:最大速度  2MHZ  1:最大速度  10MHZ
  #define PE_CR2_C0    PE_CR2,#0
  #define PE_CR2_C1    PE_CR2,#1
  #define PE_CR2_C2    PE_CR2,#2
  #define PE_CR2_C3    PE_CR2,#3
  #define PE_CR2_C4    PE_CR2,#4
  #define PE_CR2_C5    PE_CR2,#5
  #define PE_CR2_C6    PE_CR2,#6
  #define PE_CR2_C7    PE_CR2,#7

============================User_register.inc=======================
;用户定义公用变量属性说明

;00-0F 单元定义为字节变量  (供主程序使用 )
      EXTERN  R00.B     ;用户定义的变量
      EXTERN  R01.B     ;用户定义的变量
      EXTERN  R02.B     ;用户定义的变量
      EXTERN  R03.B     ;用户定义的变量
      EXTERN  R04.B     ;用户定义的变量
      EXTERN  R05.B     ;用户定义的变量
      EXTERN  R06.B     ;用户定义的变量
      EXTERN  R07.B     ;用户定义的变量
      EXTERN  R08.B     ;用户定义的变量
      EXTERN  R09.B     ;用户定义的变量
      EXTERN  R0A.B     ;用户定义的变量
      EXTERN  R0B.B     ;用户定义的变量
      EXTERN  R0C.B     ;用户定义的变量
      EXTERN  R0D.B     ;用户定义的变量
      EXTERN  R0E.B     ;用户定义的变量
      EXTERN  R0F.B     ;用户定义的变量

      EXTERN  R10.B     ;用户定义的变量
      EXTERN  R11.B     ;用户定义的变量
      EXTERN  R12.B     ;用户定义的变量
      EXTERN  R13.B     ;用户定义的变量
      EXTERN  R14.B     ;用户定义的变量
      EXTERN  R15.B     ;用户定义的变量
      EXTERN  R16.B     ;用户定义的变量
      EXTERN  R17.B     ;用户定义的变量
      EXTERN  R18.B     ;用户定义的变量
      EXTERN  R19.B     ;用户定义的变量
```

```
EXTERN  R1A.B      ;用户定义的变量
EXTERN  R1B.B      ;用户定义的变量
EXTERN  R1C.B      ;用户定义的变量
EXTERN  R1D.B      ;用户定义的变量
EXTERN  R1E.B      ;用户定义的变量
EXTERN  R1F.B     ;用户定义的变量

EXTERN  R20.B      ;用户定义的变量
EXTERN  R21.B      ;用户定义的变量
EXTERN  R22.B      ;用户定义的变量
EXTERN  R23.B      ;用户定义的变量
EXTERN  R24.B      ;用户定义的变量
EXTERN  R25.B      ;用户定义的变量
EXTERN  R26.B      ;用户定义的变量
EXTERN  R27.B      ;用户定义的变量
EXTERN  R28.B      ;用户定义的变量
EXTERN  R29.B      ;用户定义的变量
EXTERN  R2A.B      ;用户定义的变量
EXTERN  R2B.B      ;用户定义的变量
EXTERN  R2C.B      ;用户定义的变量
EXTERN  R2D.B      ;用户定义的变量
EXTERN  R2E.B      ;用户定义的变量
EXTERN  R2F.B      ;用户定义的变量

EXTERN  R30.B      ;用户定义的变量
EXTERN  R31.B      ;用户定义的变量
EXTERN  R32.B      ;用户定义的变量
EXTERN  R33.B      ;用户定义的变量
EXTERN  R34.B      ;用户定义的变量
EXTERN  R35.B      ;用户定义的变量
EXTERN  R36.B      ;用户定义的变量
EXTERN  R37.B      ;用户定义的变量
EXTERN  R38.B      ;用户定义的变量
EXTERN  R39.B      ;用户定义的变量
EXTERN  R3A.B      ;用户定义的变量
EXTERN  R3B.B      ;用户定义的变量
EXTERN  R3C.B      ;用户定义的变量
EXTERN  R3D.B      ;用户定义的变量
EXTERN  R3E.B      ;用户定义的变量
EXTERN  R3F.B     ;用户定义的变量

EXTERN  R40.W      ;用户定义的变量
EXTERN  R41.W      ;用户定义的变量
EXTERN  R42.W      ;用户定义的变量
```

```
EXTERN  R43.W     ;用户定义的变量
EXTERN  R44.W     ;用户定义的变量
EXTERN  R45.W     ;用户定义的变量
EXTERN  R46.W     ;用户定义的变量
EXTERN  R47.W     ;用户定义的变量
EXTERN  R48.W     ;用户定义的变量
EXTERN  R49.W     ;用户定义的变量
EXTERN  R4A.W     ;用户定义的变量
EXTERN  R4B.W     ;用户定义的变量
EXTERN  R4C.W     ;用户定义的变量
EXTERN  R4D.W     ;用户定义的变量
EXTERN  R4E.W     ;用户定义的变量
EXTERN  R4F.W     ;用户定义的变量
```

========== 块编程 ==========

3.3.3 块编程

Data EEPROM 和 FlashROM 均支持块编程 (块大小与芯片存储器密度有关， 如表 3-1 所示)，效率比字节、字编程要快得多，毕竟一次可同时写入一块 (64 字节或 128 字节)，所不同的是块编程要求执行编程操作的程序代码必须部分 (对于支持 RWW 功能的 Data EEPROM 至少要求数据装载程序段位于 RAM 中)、甚至全部位于 RAM 中，给程序编写带来了一定的难度。

STM8 支持标准块编程、快速块编程以及块擦除三种方式块编程，彼此之间差别不大，下面以 FlashROM 标准块编程为例，介绍块编程思路与具体实现方法。

由于块操作速度块，当写入内容超过 4 字节时，推荐用块编程方式，下面以 FlashROM 标准块编程为例，介绍块编程操作方法。

对于中高密度芯片来说， 块大小为 128B，因此需要在 RAM 空间堆栈段前使用 256 字节作为写入缓冲区与写入代码取存放区，例如：

```
IAP_First_ADR          ds.b3                    ;存放 FlashROM 块首地址
IAP_OK_Symbol          ds.b1                    ;IAP 编程成功标志 (成功时
IAP_OK_Symbol 不为 0)
    segmentbyte at D00-DFF 'ram2'               ;将 D00-DFF 定义为 ram2 段
IAP_write_data_buffer DS.B 128                  ;缓冲区大小为 128 字节
FlashROM_Block_WIRE_CODE.L  DS.B 128            ;FlashROM 块编程代码区 (必须控制在 128B 内)起始位置
```

接着在程序初始化部分将位于 FlashROM 中的块编程代码拷贝到 ram2 段中 FlashROM 块写入操作码起始位置，参考程序如下：

```
    ;编程代码转移指令系列
    LDW X, #FlashROM_Block_WIRE
    LDW Y,#FlashROM_Block_WIRE_CODE
main_FlashROM_Block_W_COPY1:
    LD A, (X)
    LD (Y), A
    CPW X, #FlashROM_Block_WIRE_END
    JREQ main_FlashROM_Block_W_COPY2
    INCW Y
```

INCW X
JRT main_FlashROM_Block_W_COPY1
main_FlashROM_Block_W_COPY2:
位于 FlashROM 标准块编程指令系列 (不能含有绝对跳转指令, 如 JP、JPF 等)如下所示。

```
;FlashROM 块编程操作子程序
;------ 入口参数 :
;                   FlashROM 块首地址在 IAP_First_ADR 中
;                   写入内容在写入缓冲区 IAP_write_data_buffer
;----- 出口参数 : 操作成功 ,IAP_OK_Symbol 单元不是 0
;----- 使用资源 :累加器 A,R00,R01 单元
BLOCK_SIZE  EQU 128                 ;块大小 (中高密度为 128B、低密度为 64B)
FlashROM_Block_WIRE:                    ;写入内容在写入缓冲区
IAP_write_data_buffer
                                    ;FlashROM 块首地址 在 IAP_First_ADR 中
    PUSHW X
    BTJT FLASH_IAPSR, #1, FlashROM_Block_write_next1
    ;0, 即状态寄存器 FLASH_IAPSR 的 PUL(b1) 位=0, FlashROM 处于保护状态, 先解除保护状态
    MOV FLASH_PUKR, #56H            ;向 FlashROM 写保护寄存器 FLASH_PUKR 连续写入 56H、AEH,
                                    ;解除写保护。
    MOV FLASH_PUKR, #0AEH
FlashROM_Block_write_next1:
    MOV IAP_OK_Symbol, #4           ;定义可重复的最大次数
    MOV FLASH_CR1, #00          ;IE 为 0, 即查询方式 ; FIX 为 0, 自动选择编程周期。
    MOV   FLASH_CR2, #01H           ;将 FLASH_CR2 寄存器的 b0 位 (PRG) 置 1,选择块编程方式
    MOV   FLASH_NCR2, #0FEH         ;将 FLASH_NCR2 寄存器的 b0 位 (NPRG) 清 0,选择块编程方式

    MOV R01, {IAP_First_ADR+2}      ;记录块首地址
FlashROM_Block_write_LOOP1:
;------ 装载 ------
    MOV R00, #BLOCK_SIZE            ;要装载的字节数
    LDW X, #IAP_write_data_buffer       ;缓冲区首地址送 Y 寄存器中
FlashROM_Block_write_LOOP11:
    LD A, (X)                       ;取操作数
    LDF [IAP_First_ADR.e],A         ;由于 Flash ROM 地址范围可能超出 FFFFH, 因此采用 24 位地址格式
    INC {IAP_First_ADR+2}           ;由于块大小只有 128B(对中高密度芯片 )或 64B(对低密度芯片 )
                                    ;即块地址单元中低 8 位为 x0000000B 或 xx000000B
```

```
        INCW X
        DEC R00
        JRNE FlashROM_Block_write_LOOP11
        ;装载结束
FlashROM_Block_write_next2:
        BTJF FLASH_IAPSR,#2, FlashROM_Block_write_next2       ;查询等待写操作
结束
;------ 校验 ------
        MOV {IAP_First_ADR+2}, R01                ;恢复块首地址
        MOV R00, #BLOCK_SIZE                      ;要校验的字节数
        LDW X, #IAP_write_data_buffer             ;缓冲区首地址送 X 寄存器中
FlashROM_Block_write_LOOP12:
        LDF A, [IAP_First_ADR.e]
        XOR A, (X)
        JRNE FlashROM_Block_write_next3
        ;本单元校验正确
        INC {IAP_First_ADR+2}                     ;由于块大小只有 128B(对中高密度
芯片)或 64B(对低密度
                                                  ;芯片)即块地址单元中低 8 位为 x0000000B
或 xx000000B
        INCW X
        DEC R00
        JRNE FlashROM_Block_write_LOOP12
        ;整个模块校验正确
        MOV {IAP_First_ADR+2}, R01                ;恢复块首地址
        JRT FlashROM_Block_write_exit; 校验正确！
FlashROM_Block_write_next3:
        ;校验错误，恢复块收地址后重新装入，再写
        MOV {IAP_First_ADR+2}, R01                ;恢复块首地址
        DEC IAP_OK_Symbol
        JRNE FlashROM_Block_write_LOOP1
FlashROM_Block_write_exit:
        BRES FLASH_IAPSR,#1                       ;清除 PUL 位,恢复写保护状态
        POPW X                                    ;恢复 X 内容
        RETF
FlashROM_Block_WIRE_END:
RETF
这样就可以按照如下方式调用，执行标准块写入操作
LDW X, #IAP_write_data_buffer
LD (X), #XXH                                      ;初始化缓冲区
MOV {IAP_First_ADR+0}, #XXH                       ;初始化块首地址高 16 位
MOV {IAP_First_ADR+1}, #XXH                       ;初始化块首地址高 8 位
MOV {IAP_First_ADR+2}, #XXH                       ;初始化块首地址低 8 位
```

CALLF FlashROM_Block_WIRE_CODE ;RAM2 段中定义的标准块编程代码首地址标号

如果每次写入内容小于块容量时, 可采用读 (把整快读入缓冲区内)→改 (改写指定的指定字节)→写 (再写入 FlashROM 对应块)方式进行。

==========stm8==========

STM8 与汇编语言（ 1）

不知是心血来潮, 还是其它因素, 突然又想起玩汇编语言了。 这几年也没少跟单片机打交道，包括 51 系列， 430 系列， ARM 系列，但都是用 C 语言来开发。不过由于使用 C 语言，实际上对这些 CPU 的了解还是不够深刻，当然除了 51 之外，因为那是我多年前曾经用汇编开发过的芯片。尽管当今 C 语言已经在嵌入式产品的开发过程中成为主流, 但我个人依然认为, 要想真正了解 CPU 的特点，还得用汇编语言。不知道这种观点是对还是错，也许是因为自己从硬件做起, 写过机器码，用汇编语言做过优化，因此对汇编语音有一种特殊的偏爱。
51 系列的芯片用多了，感觉有时写起程序来不太方便，因此总想寻找一些其它的 8 位单片机玩玩, 正好手头有一个 ST 的三合一开发板, 那是 09 年参加 ST 研讨会上买的，一直躺在那里，与其躺在那里，不如拿出来玩玩。
这几年, ST 在国内推广 STM32，力度不小，不过我一直没有用过，只是初步地看看资料。原因在于在 32 位单片机方面，我一直在用 Luminary 公司的 LM3S1138，感觉不错，一直都很顺利。 09 年 ST 举办的研讨会上, ST 除了介绍 STM32 外，也介绍了 STM8，当时听了以后，觉得还行。尤其是会上的低功耗演示给我留下了很深刻的印象。
基于这些，我决定好好地玩一下 STM8 芯片，并将玩的结果拿出来与大家共享。

STM8 与汇编语言（ 2）

第一次打开 STM8 的手册时发现, CPU 中的寄存器只有 6 个，即 A、X、Y、SP、PC 和 CC。这几个寄存器，看上去特象早年苹果机使用的微处理器 6502。在眼下都是多寄存器的 RISC 潮流下，不知 ST 推出的这种 CPU 架构有什么意图？这样的芯片能否与 Microchip 或者 Atmel 的 RISC 结构的 MCU 竞争呢？在此我无意做评论，我只想了解这颗芯片。
通过仔细研究，我发现由于 STM8 采用了 32 位宽度的程序存储器结构，使得大部分的指令都能在一个周期内取出, 并且采用了哈佛结构和流水线, 相当多的指令也都是单周期完成的。这样的话，虽然 CPU 是 CISC 架构的，但也基本上达到了单周期指令的效果, 就像手册上说的, CPU 的性能达到了 20MISP◎24MHZ。就这一点来说，我个人感觉 STM8 还真不错。
举个例子来说, 如果我们要完成内存中的 2 个 8 位无符号数相加, 结果还保存到内存中，用 C 语言描述成：

```
unsigned char a,b,c;
c = a + b;
```

这一段程序，用 STM8 汇编可以写成如下代码：

```
LD      A, $1000
ADD     A, $1001
LD      $1002, A
```

这里假设 a,b,c这 3 个变量分别存储在内存中， 地址为 1000,1001和 1002。从 STM8 的手册上可以查到， 这 3 条指令都是单周期的， 完成一个加法， 只需要 3 个时钟周期，可见 STM8 的 CPU 执行速度还是相当快的。

在这种传统的所谓 CISC 架构中， 我特别关心累加器 A 与内存的访问速度， 因为如果累加器与内存的访问速度是单周期的话， 实际上我们就可以将内存当寄存器来看，这样写程序时就相当于拥有了一个大的寄存器阵列， 或者说我们也就没必要再去考虑变量在内存中还是在寄存器中。 也正是因为这一点， 我对 STM8 越来越感兴趣了。

STM8 与汇编语言（ 3）

STM8 的开发环境用起来还是不错的，可以到 ST 的网站上下载安装程序 ST_Toolset.exe。利用该环境可以开发用汇编语言写的程序， 而且与 ST 的三合一开发板配合起来，确实非常方便。

不过如果要想用 C 语言来开发，稍微有点麻烦，得去别的公司下载 C 的编译器（CXSTM8_16K.exe），而且下载完以后，还得去注册，等待许可文件。实际上，我也按照 ST 介绍的方法做了， 但始终都没有收到许可文件， 也许本人实在愚笨。但不管怎么说， 我觉得 ST 这一点做得相当不好， 实在有点抠门。 既然是免费的，为什么不一起打包提供给客户，这么麻烦，多耽误客户使用，得少卖多少 STM8 的芯片。

言归正传，还回到正题。用汇编语言开发程序，最简单的就是利用 ST 开发环境中提供的汇编程序框架自动生成功能。 打开开发环境后， 在 File 菜单中选择 New Workspace,点击 Create workspace and project 图标，然后就可以建立项目， 在工具链中选 ST Assembler Linker，最后选择 MCU 的型号，点击 OK，就完成了一个项目的建立。这个环境与微软的 VC6 开发环境很象，点开项目文件中的 Source Files，能看到系统自动生成好了一个汇编语言的框架， 我们编写程序只要在这框架基础上就可以了。 其实不用编写任何一条指令， 这个框架程序是能够编译通过，并下载运行的。

自动生成的项目中包含 3 个重要的文件： mapping.inc,mapping.asm和 main.asm。mapping.inc 文件中定义的是一些常量， mapping.asm文件中定义的是一些内存的分配，主要的汇编代码都在 main.asm。

下面是 main.asm中的汇编代码及注释。

```
stm8/
        #include "mapping.inc"
        segment'rom'
;  下面是定义一个标号，  ST 汇编的写法，有点不习惯
;  这里的 main 标号是复位后的第一条指令，与后面的中断向量表中
;  的名字是对应的
main.l
        ; initialize  SP
```

```
        ldw  X,#stack_end
        ldw  SP,X                                   ;  设置堆栈指针

        #ifdef  RAM0
;  如果定义了   RAM0 ,  则要汇编以下代码
        ; clear RAM0
ram0_start.b EQU $ram0_segment_start
ram0_end.b EQU $ram0_segment_end
        ldw  X,#ram0_start      ;寄存器  X 指向要清除的内存起始地址
clear_ram0.l                ;这是一个标号定义，用于后面的跳转指令
        clr  (X)                  ;对应的内存单元清    0
        incw X                    ;寄存器  X+1,指向下一个单元
        cpw X,#ram0_end           ;比较寄存器   X 是否等于内存的最后一个地址
        jrule clear_ram0          ;若不等于，则循环
        #endif

        #ifdef  RAM1
;  如果定义了   RAM1 ,  则要汇编以下代码，代码含义与上面完全一样
        ; clear RAM1
ram1_start.w EQU $ram1_segment_start
ram1_end.w EQU $ram1_segment_end
        ldw  X,#ram1_start
clear_ram1.l
        clr  (X)
        incw X
        cpw X,#ram1_end
        jrule clear_ram1
        #endif

        ; clear stack
;  将堆栈区的内存单元清    0，代码含义与上面完全一样
stack_start.w EQU $stack_segment_start
stack_end.w EQU $stack_segment_end
        ldw  X,#stack_start
clear_stack.l
        clr  (X)
        incw X
        cpw X,#stack_end
        jrule clear_stack

infinite_loop.l                             ;  定义一个标号
;由于是一个框架，初始化内存后，进入一个死循环
        jra infinite_loop
```

```
;下面代码写的是一段中断服务程序，不过也是空的
        interrupt NonHandledInterrupt
NonHandledInterrupt.l
;  当进入中断服务程序后，无其它动作，直接返回
   iret

;  下面这张表很重要，定义了    STM8 所有的硬件中断对应的中断
;  服务程序的入口地址
   segment'vectit'
   dc.l {$82000000+main}                    ; reset
   dc.l {$82000000+NonHandledInterrupt}     ; trap
   dc.l {$82000000+NonHandledInterrupt}     ; irq0
   dc.l {$82000000+NonHandledInterrupt}     ; irq1
   dc.l {$82000000+NonHandledInterrupt}     ; irq2
   dc.l {$82000000+NonHandledInterrupt}     ; irq3
   dc.l {$82000000+NonHandledInterrupt}     ; irq4
   dc.l {$82000000+NonHandledInterrupt}     ; irq5
   dc.l {$82000000+NonHandledInterrupt}     ; irq6
   dc.l {$82000000+NonHandledInterrupt}     ; irq7
   dc.l {$82000000+NonHandledInterrupt}     ; irq8
   dc.l {$82000000+NonHandledInterrupt}     ; irq9
   dc.l {$82000000+NonHandledInterrupt}     ; irq10
   dc.l {$82000000+NonHandledInterrupt}     ; irq11
   dc.l {$82000000+NonHandledInterrupt}     ; irq12
   dc.l {$82000000+NonHandledInterrupt}     ; irq13
   dc.l {$82000000+NonHandledInterrupt}     ; irq14
   dc.l {$82000000+NonHandledInterrupt}     ; irq15
   dc.l {$82000000+NonHandledInterrupt}     ; irq16
   dc.l {$82000000+NonHandledInterrupt}     ; irq17
   dc.l {$82000000+NonHandledInterrupt}     ; irq18
   dc.l {$82000000+NonHandledInterrupt}     ; irq19
   dc.l {$82000000+NonHandledInterrupt}     ; irq20
   dc.l {$82000000+NonHandledInterrupt}     ; irq21
   dc.l {$82000000+NonHandledInterrupt}     ; irq22
   dc.l {$82000000+NonHandledInterrupt}     ; irq23
   dc.l {$82000000+NonHandledInterrupt}     ; irq24
   dc.l {$82000000+NonHandledInterrupt}     ; irq25
   dc.l {$82000000+NonHandledInterrupt}     ; irq26
   dc.l {$82000000+NonHandledInterrupt}     ; irq27
   dc.l {$82000000+NonHandledInterrupt}     ; irq28
   dc.l {$82000000+NonHandledInterrupt}     ; irq29

   end
```

把这个项目 Build 后，点击 Debug 中的 Start Debugging 就可以将程序下载到 ST 的三合一板上了，然后点击 Run，程序就运行起来了，不过由于框架程序是一个空程序， 初始化内存后就进入死循环了， 因此什么效果也看不见。 因此我们必须在框架程序的基础上， 编写自己的程序。 后面的程序例子都是在这个框架程序的基础上编写的。

STM8 与汇编语言（ 4）

今天要做的实验是在 ST 的三合一开发板上，用汇编语言写一个程序，驱动板上的 LED 指示灯闪烁。
开发板上的 LED1 接在 STM8 的 PD3 上，因此要将 PD3 设置成输出模式，为了提高高电平时的输出电流， 要将其设置成推挽输出方式。 这主要通过设置对应的 DDR/CR1/CR2 寄存器实现。
还是利用 ST 的开发工具，先生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。
编译通过后，下载到开发板，运行程序，可以看到 LED1 在闪烁，且闪烁的频率为 5HZ→1 秒闪 5 下。

stm8/

```
        #include "mapping.inc"

; 下面定义端口 D 的寄存器地址
PD_ODR  EQU   $500f
PD_IDR  EQU   $5010
PD_DDR  EQU   $5011
PD_CR1  EQU   $5012
PD_CR2  EQU   $5013

; 定义堆栈空间的起始位置和结束位置
stack_start.w EQU   $stack_segment_start
stack_end.w   EQU   $stack_segment_end

; 下面开始定义一个段，该段位于 ROM 中
              segment'rom'
; 定义复位后的第一条指令的标号（即入口地址）
main.l
;
; 首先要初始化堆栈指针
              LDW   X,#stack_end
              LDW   SP,X
```

```
        LD      A,#08
        LD      PD_DDR,A            ; 将 PD3 设置成输出
        LD      A,#08
        LD      PD_CR1,A            ; 将 PD3 设置成推挽输出
        LD      A,#00
        LD      PD_CR2,A            ;
MAIN_LOOP.L
        LD      A,#08               ;
        LD      PD_ODR,A            ; 将 PD3 的输出设置成 1
        LD      A,#100              ;为什么是 100？？？
        CALL    DELAY_MS            ; 调用自定义延时函数, 延时 100MS

      ; 输出参数：无
; 返 回 值：无
; 备 注：无
DELAY_MS.L
        PUSH    A       ; 将入口参数保存到堆栈中     #100 压入堆栈
        LD      A,#250              ; 寄存器 A<-250,作为下面的循环数
DELAY_MS_1.L
        NOP                         ; 用空操作指令进行延时 4T
        NOP
        NOP
        NOP
        NOP
        NOP
        DEC     A                   ; 寄存器 A<-A-1，本条指令执行之间为 1T
        JRNE    DELAY_MS_1          ; 若不等于 0，则循环,
                                    ; 本条指令执行时间为
2T（跳时）或 1T（不跳时）
        POP     A                   ; 从堆栈中恢复入口参数
        DEC     A                   ; 将要延时的 MS 数－1
        JRNE    DELAY_MS            ; 若不等于 0, 则循环
        RET                         ; 函数返回
;CALL 指令执行后，什么时候调用结束，返回的标志是什么，是不是     RET?是
RET

  interrupt NonHandledInterrupt
NonHandledInterrupt.l
        iret

; 下面定义中断向量表
        segment'vectit'
        dc.l   {$82000000+main}                         ; reset
        dc.l {$82000000+NonHandledInterrupt}    ; trap
        dc.l {$82000000+NonHandledInterrupt}    ; irq0
```

```
dc.l {$82000000+NonHandledInterrupt}      ; irq1
dc.l {$82000000+NonHandledInterrupt}      ; irq2
dc.l {$82000000+NonHandledInterrupt}      ; irq3
dc.l {$82000000+NonHandledInterrupt}      ; irq4
dc.l {$82000000+NonHandledInterrupt}      ; irq5
dc.l {$82000000+NonHandledInterrupt}      ; irq6
dc.l {$82000000+NonHandledInterrupt}      ; irq7
dc.l {$82000000+NonHandledInterrupt}      ; irq8
dc.l {$82000000+NonHandledInterrupt}      ; irq9
dc.l {$82000000+NonHandledInterrupt}      ; irq10
dc.l {$82000000+NonHandledInterrupt}      ; irq11
dc.l {$82000000+NonHandledInterrupt}      ; irq12
dc.l {$82000000+NonHandledInterrupt}      ; irq13
dc.l {$82000000+NonHandledInterrupt}      ; irq14
dc.l {$82000000+NonHandledInterrupt}      ; irq15
dc.l {$82000000+NonHandledInterrupt}      ; irq16
dc.l {$82000000+NonHandledInterrupt}      ; irq17
dc.l {$82000000+NonHandledInterrupt}      ; irq18
dc.l {$82000000+NonHandledInterrupt}      ; irq19
dc.l {$82000000+NonHandledInterrupt}      ; irq20
dc.l {$82000000+NonHandledInterrupt}      ; irq21
dc.l {$82000000+NonHandledInterrupt}      ; irq22
dc.l {$82000000+NonHandledInterrupt}      ; irq23
dc.l {$82000000+NonHandledInterrupt}      ; irq24
dc.l {$82000000+NonHandledInterrupt}      ; irq25
dc.l {$82000000+NonHandledInterrupt}      ; irq26
dc.l {$82000000+NonHandledInterrupt}      ; irq27
dc.l {$82000000+NonHandledInterrupt}      ; irq28
dc.l {$82000000+NonHandledInterrupt}      ; irq29


        end
```

STM8 与汇编语言（ 5）

上一次的实验程序，完成了 LED 指示灯的驱动，用到了 GPIO 的输出方式，这一次要用 GPIO 的输入方式，进行按键的输入。下面的代码是读入按键值，如果按键按下，则点亮 LED，否则熄灭 LED。

利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。

编译通过后，下载到开发板，运行程序，可以看到当按下按键时， LED1 点亮，当抬起按键时， LED1 熄灭。

stm8/

```
        #include "mapping.inc"

; 涉及到的硬件资源
; LED1 定义在 PD3
; KEY1 定义在 PD7

; 下面定义端口 D 的寄存器地址
PD_ODR   EQU    $500f
PD_IDR   EQU    $5010
PD_DDR   EQU    $5011
PD_CR1   EQU    $5012
PD_CR2   EQU    $5013


; 定义堆栈空间的起始位置和结束位置
stack_start.w  EQU    $stack_segment_start
stack_end.w    EQU    $stack_segment_end


                segment'rom'          ; 下面开始定义一个段，该段位于 ROM 中
main.l                                ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
        LDW     X,#stack_end
        LDW     SP,X


; 下面初始化 IO 端口
; PD3 设置成推挽输出
; PD7 设置成悬浮输入
        LD      A,#08
  LD      PD_DDR,A              ; 将 PD3 设置成输出， PD7 设置成输入
        LD      A,#08
        LD      PD_CR1,A              ; 将 PD3 设置成推挽输出
        LD      A,#00
        LD      PD_CR2,A              ;
MAIN_LOOP.L
```

```
        LD      A,PD_IDR            ; 读入端口 D 的引脚输入寄存器
        AND     A,#$80              ; 测试最高位是否为 1
        JRNE    MAIN_LOOP_1                 ; 若最高位为 1,
则跳转
        LD      A,#$08              ; 否则说明按键按下,  PD3<-1,点亮 LED1
        LD      PD_ODR,A            ;
        JRA     MAIN_LOOP
MAIN_LOOP_1.L
        LD      A,#$00              ; 若按键没按下,  PD3<-0,熄灭 LED1
        LD      PD_ODR,A            ;
        JRA     MAIN_LOOP           ;
LD      A,#00                       ;
        LD      PD_ODR,A            ; 将 PD3 的输出设置成 1
        LD      A,#100
        CALL    DELAY_MS            ; 延时 100MS

        JRA     MAIN_LOOP           ;


; 函数功能：延时
; 输入参数：寄存器 A ——  要延时的毫秒数，这里假设  CPU 的主频为 2MHZ


  interrupt NonHandledInterrupt
NonHandledInterrupt.l
                                    iret


; 下面定义中断向量表
                segment'vectit'
                dc.l {$82000000+main}                       ; reset
                dc.l {$82000000+NonHandledInterrupt}    ; trap
                dc.l {$82000000+NonHandledInterrupt}    ; irq0
                dc.l {$82000000+NonHandledInterrupt}    ; irq1
                dc.l {$82000000+NonHandledInterrupt}    ; irq2
                dc.l {$82000000+NonHandledInterrupt}    ; irq3
                dc.l {$82000000+NonHandledInterrupt}    ; irq4
                dc.l {$82000000+NonHandledInterrupt}    ; irq5
                dc.l {$82000000+NonHandledInterrupt}    ; irq6
                dc.l {$82000000+NonHandledInterrupt}    ; irq7
                dc.l {$82000000+NonHandledInterrupt}    ; irq8
                dc.l {$82000000+NonHandledInterrupt}    ; irq9
                dc.l {$82000000+NonHandledInterrupt}    ; irq10
```

```
dc.l {$82000000+NonHandledInterrupt}      ; irq11
dc.l {$82000000+NonHandledInterrupt}      ; irq12
dc.l {$82000000+NonHandledInterrupt}      ; irq13
dc.l {$82000000+NonHandledInterrupt}      ; irq14
dc.l {$82000000+NonHandledInterrupt}      ; irq15
dc.l {$82000000+NonHandledInterrupt}      ; irq16
dc.l {$82000000+NonHandledInterrupt}      ; irq17
dc.l {$82000000+NonHandledInterrupt}      ; irq18
dc.l {$82000000+NonHandledInterrupt}      ; irq19
dc.l {$82000000+NonHandledInterrupt}      ; irq20
dc.l {$82000000+NonHandledInterrupt}      ; irq21
dc.l {$82000000+NonHandledInterrupt}      ; irq22
dc.l {$82000000+NonHandledInterrupt}      ; irq23
dc.l {$82000000+NonHandledInterrupt}      ; irq24
dc.l {$82000000+NonHandledInterrupt}      ; irq25
dc.l {$82000000+NonHandledInterrupt}      ; irq26
dc.l {$82000000+NonHandledInterrupt}      ; irq27
dc.l {$82000000+NonHandledInterrupt}      ; irq28
dc.l {$82000000+NonHandledInterrupt}      ; irq29


end
```

STM8 与汇编语言（ 6）－－ 8 位定时器应用之一

STM8 单片机中的外设资源是比较丰富的，定时器有 8 位的也有 16 位的，下面的实验程序，就是利用 8 位定时器 4 来进行延时，然后驱动 LED 闪烁。
同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。
编译通过后，下载到开发板，运行程序，可以看到 LED 在闪烁，或者用示波器可以在 LED 引脚上看到方波。
在这里要特别提醒的是，从 ST 给的手册上看，这个定时器中的计数器是一个加 1 计数器，但本人在实验过程中感觉不太对，经过反复的实验，我认为应该是一个减 1 计数器（也许是我拿的手册不对，或许是理解上有误） 。例如，当给定时器中的自动装载寄存器装入 255 时，产生的方波频率最小， 就象下面代码中计算的那样，产生的方波频率为 30HZ 左右。若初始化时给自动装载寄存器装入 1，则产生的方波频率最大， 大约为 3.9K 左右。也就是说实际的分频数为 ARR 寄存器的值 +1。

stm8/

```
        #include "mapping.inc"

;       #include "STM8S207S8.INC"


; 涉及到的硬件资源
; 下面定义端口  D 的寄存器地址
PD_ODR   EQU    $500f
PD_IDR   EQU    $5010
PD_DDR   EQU    $5011
PD_CR1   EQU    $5012
PD_CR2   EQU    $5013


; 定时器  4 的寄存器定义
TIM4_CR1    EQU    $5340
TIM4_IER    EQU    $5341
TIM4_SR     EQU    $5342
TIM4_EGR    EQU    $5343
TIM4_CNTR   EQU    $5344
TIM4_PSCR   EQU    $5345
TIM4_ARR    EQU    $5346


; 定义堆栈空间的起始位置和结束位置
stack_start.w  EQU    $stack_segment_start
stack_end.w    EQU    $stack_segment_end



                segment'rom'            ; 下面开始定义一个段，该段位于  ROM 中
main.l                                  ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
                LDW     X,#stack_end
                LDW     SP,X


; 下面初始化  IO 端口
; PD3 设置成推挽输出
; PD7 设置成悬浮输入
                LD      A,#08
                LD      PD_DDR,A        ; 将 PD3 设置成输出，  PD7 设置成输入
                LD      A,#08
```

```
                LD      PD_CR1,A        ; 将 PD3 设置成推挽输出
              LD      A,#00
                    LD      PD_CR2,A                ;
;
; 下面初始化定时器  4
              LD      A,#$00
               LD      TIM4_IER,A        ; 禁止中断
              LD      A,#$01
              LD      TIM4_EGR,A        ; 允许产生更新事件
                    LD      A,#$07
                    LD      TIM4_PSCR,A       ; 计数器时钟 =主时钟 /128=2MHZ/128
                                              ; 相当于计数器周期为   64uS
                    LD      A,#255
                    LD      TIM4_ARR,A        ; 设定重装载时的寄存器值,   255 是最
大值
                    LD      A,#255
                    LD      TIM4_CNTR,A       ; 设定计数器的初值
                                              ; 定时周期
=(ARR+1)*64=16384uS
                                          ; 产生方波频率 =30.5HZ

                                          ;
LD      A,#$01
                                              ; b0 = 1,允许计数器工作
; b1 = 0,允许更新
LD      TIM4_CR1,A        ; 设置控制器，启动定时器
MAIN_LOOP.L
                LD      A,TIM4_SR             ; 读入定时器  4 的状态
                AND     A,#01                 ; 判断是否产生更新标志
                JREQ    MAIN_LOOP             ; 若没有，则等待
         LD      A,#0                     ; 清除更新标志
         LD      TIM4_SR,A

        LD      A,PD_ODR                  ; 将 LED 驱动信号取反
        XOR     A,#$08
        LD      PD_ODR,A                  ; LED 闪烁频率 =2MHZ/128/255/2=30.63
        JRA     MAIN_LOOP                 ; 无限循环



   interrupt NonHandledInterrupt
NonHandledInterrupt.l
          iret
```

```
; 下面定义中断向量表
                segment'vectit'
                dc.l
{$82000000+main}                                                    ; reset
                dc.l {$82000000+NonHandledInterrupt}      ; trap
                dc.l {$82000000+NonHandledInterrupt}      ; irq0
                dc.l {$82000000+NonHandledInterrupt}      ; irq1
                dc.l {$82000000+NonHandledInterrupt}      ; irq2
                dc.l {$82000000+NonHandledInterrupt}      ; irq3
                dc.l {$82000000+NonHandledInterrupt}      ; irq4
                  dc.l {$82000000+NonHandledInterrupt}      ; irq5
                dc.l {$82000000+NonHandledInterrupt}      ; irq6
                dc.l {$82000000+NonHandledInterrupt}      ; irq7
                dc.l {$82000000+NonHandledInterrupt}      ; irq8
                dc.l {$82000000+NonHandledInterrupt}      ; irq9
                dc.l {$82000000+NonHandledInterrupt}      ; irq10
                dc.l {$82000000+NonHandledInterrupt}      ; irq11
                dc.l {$82000000+NonHandledInterrupt}      ; irq12
                dc.l {$82000000+NonHandledInterrupt}      ; irq13
                dc.l {$82000000+NonHandledInterrupt}      ; irq14
                dc.l {$82000000+NonHandledInterrupt}      ; irq15
                dc.l {$82000000+NonHandledInterrupt}      ; irq16
                dc.l {$82000000+NonHandledInterrupt}      ; irq17
                dc.l {$82000000+NonHandledInterrupt}      ; irq18
                dc.l {$82000000+NonHandledInterrupt}      ; irq19
                dc.l {$82000000+NonHandledInterrupt}      ; irq20
                dc.l {$82000000+NonHandledInterrupt}      ; irq21
                dc.l {$82000000+NonHandledInterrupt}      ; irq22
                dc.l {$82000000+NonHandledInterrupt}      ; irq23
                dc.l {$82000000+NonHandledInterrupt}      ; irq24
                dc.l {$82000000+NonHandledInterrupt}      ; irq25
                dc.l {$82000000+NonHandledInterrupt}      ; irq26
                dc.l {$82000000+NonHandledInterrupt}      ; irq27
                dc.l {$82000000+NonHandledInterrupt}      ; irq28
                  dc.l {$82000000+NonHandledInterrupt}      ; irq29

        end
```

STM8 与汇编语言（ 7）－－ 8 位定时器应用之二

上次写的是用 STM8 单片机中的 8 位定时器作为软件延时，采用的是查询方式。在实际系统中，定时器的应用，更多的是采用中断方式，下面的代码就给出 8 位定时器在中断方式下的应用。

实验程序首先初始化驱动 LED 的端口，然后初始化 8 位的定时器 4，最后启动中断允许，要记住，一定要将中断服务程序的入口地址填写到中断向量表中，并且要根据中断向量号在正确的位置上填写。

同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。

编译通过后，下载到开发板，运行程序，可以看到 LED 在闪烁，或者用示波器可以在 LED 引脚上看到方波。

stm8/

```
        #include "mapping.inc"

;       #include "STM8S207S8.INC"


;  涉及到的硬件资源
; LED1 定义在 PD3

;  下面定义端口 D 的寄存器地址
PD_ODR    EQU     $500f
PD_IDR    EQU     $5010
PD_DDR    EQU     $5011
PD_CR1    EQU     $5012
PD_CR2    EQU     $5013


;  定时器 4 的寄存器定义
TIM4_CR1    EQU     $5340
TIM4_IER    EQU     $5341
TIM4_SR     EQU     $5342
TIM4_EGR    EQU     $5343
TIM4_CNTR   EQU     $5344
TIM4_PSCR   EQU     $5345
TIM4_ARR    EQU     $5346
```

```
; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU      $stack_segment_start
stack_end.w     EQU      $stack_segment_end


                segment'rom'                ; 下面开始定义一个段，该段位于     ROM 中
main.l                                      ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
                LDW      X,#stack_end
                LDW      SP,X


; 下面初始化   IO 端口
; PD3 设置成推挽输出
; PD7 设置成悬浮输入
                LD       A,#08
                LD       PD_DDR,A           ; 将 PD3 设置成输出，   PD7 设置成输入
                LD       A,#08
                LD       PD_CR1,A           ; 将 PD3 设置成推挽输出
                LD       A,#00
                LD       PD_CR2,A           ;
;
; 下面初始化定时器   4
                LD       A,#$01
                LD       TIM4_EGR,A         ; 允许产生更新事件
                LD       A,#$07
                LD       TIM4_PSCR,A        ; 计数器时钟 =主时钟 /128=2MHZ/128
                                                    ; 相当于计数 器周期为
64uS
                LD       A,#255
                LD       TIM4_ARR,A         ; 设定重装载时的寄存器值，     255 是最大值

                LD       A,#255
                LD       TIM4_CNTR,A        ; 设定计数器的初值
                                            ; 定时周期 =(255+1)*64=16384uS


LD              A,#$01                  ; b0 = 1,允许计数器工作
                                            ; b1 = 0,允许更新


LD              TIM4_CR1,A              ; 设置控制器，启动定时器
```

```
        LD      A,#$01          ; 允许更新中断
    LD      TIM4_IER,A      ;
        RIM                     ; 允许 CPU 全局中断

MAIN_LOOP.L
        JRA     MAIN_LOOP       ; 进入无限循环


; 下面是定时器  4 的中断服务程序
TIMER4_ISR.L
        LD      A,#0            ; 清除更新标志
        LD      TIM4_SR,A
        LD      A,PD_ODR        ; 将 LED 驱动信号取反
        XOR     A,#$08
        LD      PD_ODR,A        ; LED 闪烁频率 =2MHZ/128/256/2=30.5
        IRET                    ; 中断返回



  interrupt NonHandledInterrupt
NonHandledInterrupt.l
                                iret

; 下面定义中断向量表
                segment'vectit'
                dc.l {$82000000+main}                   ; reset
                dc.l {$82000000+NonHandledInterrupt}    ; trap
                 dc.l {$82000000+NonHandledInterrupt}   ; irq0
                 dc.l {$82000000+NonHandledInterrupt}   ; irq1
                 dc.l {$82000000+NonHandledInterrupt}   ; irq2
                 dc.l {$82000000+NonHandledInterrupt}   ; irq3
                 dc.l {$82000000+NonHandledInterrupt}   ; irq4
                 dc.l {$82000000+NonHandledInterrupt}   ; irq5
                 dc.l {$82000000+NonHandledInterrupt}   ; irq6
                dc.l {$82000000+NonHandledInterrupt}    ; irq7
                dc.l {$82000000+NonHandledInterrupt}    ; irq8
                dc.l {$82000000+NonHandledInterrupt}    ; irq9
                dc.l {$82000000+NonHandledInterrupt}    ; irq10
                dc.l {$82000000+NonHandledInterrupt}    ; irq11
                dc.l {$82000000+NonHandledInterrupt}    ; irq12
                dc.l {$82000000+NonHandledInterrupt}    ; irq13
                dc.l {$82000000+NonHandledInterrupt}    ; irq14
                dc.l {$82000000+NonHandledInterrupt}    ; irq15
```

```
dc.l {$82000000+NonHandledInterrupt}      ; irq16
dc.l {$82000000+NonHandledInterrupt}      ; irq17
dc.l {$82000000+NonHandledInterrupt}      ; irq18
dc.l {$82000000+NonHandledInterrupt}      ; irq19
dc.l {$82000000+NonHandledInterrupt}      ; irq20
dc.l {$82000000+NonHandledInterrupt}      ; irq21
dc.l {$82000000+NonHandledInterrupt}      ; irq22
dc.l {$82000000+TIMER4_ISR}                    ; irq23
                                  ; 对应的是定时器 4 的中断入口
        dc.l {$82000000+NonHandledInterrupt}      ; irq24
        dc.l {$82000000+NonHandledInterrupt}      ; irq25
        dc.l {$82000000+NonHandledInterrupt}      ; irq26
        dc.l {$82000000+NonHandledInterrupt}      ; irq27
        dc.l {$82000000+NonHandledInterrupt}      ; irq28
        dc.l {$82000000+NonHandledInterrupt}      ; irq29


    end
```

STM8 与汇编语言（ 8）－－ 16 位定时器应用

当需要更长时间的定时时， 最好使用 16 位的定时器， STM8 单片机中都提供了 2 到 3 个的 16 位定时器，方便用户使用。

下面的代码给出了一个采用 16 位定时器实现的定时中断程序，在定时中断程序中，驱动 LED 指示灯的闪烁。

切记，一定要将中断服务程序的入口地址填写到中断向量表中， 并且要根据定时器的中断向量号在正确的位置上填写。

同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。

编译通过后，下载到开发板，运行程序，可以看到 LED 在闪烁，或者用示波器可以在 LED 引脚上看到方波。

```
stm8/

    #include "mapping.inc"

    #include "STM8S207C_S.INC"

; 涉及到的硬件资源
```

```
; LED1 定义在  PD3
; KEY1  定义在  PD7



;  定义堆栈空间的起始位置和结束位置
stack_start.w   EQU       $stack_segment_start
stack_end.w     EQU       $stack_segment_end
                segment'rom'            ;  下面开始定义一个段，该段位于     ROM 中
main.l                                  ;  定义复位后的第一条指令的标号（即入口地址）
;
;  首先要初始化堆栈指针
                LDW       X,#stack_end
                LDW       SP,X


;  下面初始化   IO 端口
; PD3 设置成推挽输出
; PD7 设置成悬浮输入
                LD        A,#08
                LD        PD_DDR,A      ;  将 PD3 设置成输出，    PD7 设置成输入
                LD        A,#08
                LD        PD_CR1,A      ;  将 PD3 设置成推挽输出
                LD        A,#00
                LD        PD_CR2,A
;
;  下面初始化定时器   4
                LD        A,#$01
                LD        TIM2_EGR,A    ;  允许产生更新事件
                LD        A,#$01
                LD        TIM2_PSCR,A ;  计数器时钟  =主时钟 /2=2MHZ/2
                                        ;  相当于计数器周期为    1uS


;  设定重装载时的寄存器值
;  注意必须保证先写入高    8 位，再写入低   8 位
                LD        A,#$ea
                LD        TIM2_ARRH,A   ;  设定重装载时的寄存器值的高     8 位
                LD        A,#$60
                LD        TIM2_ARRL,A   ;  设定重装载时的寄存器值的低     8 位
```

```
LD      A,#$ea
LD      TIM2_CNTRH,A;   设定计数器的初值的高   8 位
LD      A,#$60
LD      TIM2_CNTRL,A;   设定计数器的初值的低   8 位
                        ;  定时周期 =60000*1=60mS
LD      A,#$01
                ; b0 = 1,允许计数器工作
                ; b1 = 0,允许更新
    LD      TIM2_CR1,A   ;  设置控制器，启动定时器
        LD      A,#$01      ;  允许更新中断
        LD      TIM2_IER,A
        RIM                 ;  允许  CPU 全局中断

MAIN_LOOP.L
        JRA     MAIN_LOOP   ;  进入无限循环


;  下面是定时器   2 的中断服务程序
Timer2_Update_ISR.L
        LD      A,#0        ;  清除更新标志
        LD      TIM2_SR1,A
        LD      A,PD_ODR    ;  将 LED  驱动信号取反
        XOR     A,#$08
        LD      PD_ODR,A    ; LED  闪烁频率 =2MHZ/2/60000/2=8.3HZ
        IRET                ;  中断返回


  interrupt NonHandledInterrupt
NonHandledInterrupt.l
                                iret


;  下面定义中断向量表
segment'vectit'
dc.l {$82000000+main}                                   ; reset
dc.l {$82000000+NonHandledInterrupt}    ; trap
dc.l {$82000000+NonHandledInterrupt}    ; irq0
dc.l {$82000000+NonHandledInterrupt}    ; irq1
dc.l {$82000000+NonHandledInterrupt}    ; irq2
dc.l {$82000000+NonHandledInterrupt}    ; irq3
dc.l {$82000000+NonHandledInterrupt}    ; irq4
dc.l {$82000000+NonHandledInterrupt}    ; irq5
```

```
dc.l {$82000000+NonHandledInterrupt}      ; irq6
dc.l {$82000000+NonHandledInterrupt}      ; irq7
dc.l {$82000000+NonHandledInterrupt}      ; irq8
dc.l {$82000000+NonHandledInterrupt}      ; irq9
dc.l {$82000000+NonHandledInterrupt}      ; irq10
dc.l {$82000000+NonHandledInterrupt}      ; irq11
dc.l {$82000000+NonHandledInterrupt}      ; irq12
dc.l {$82000000+Timer2_Update_ISR}          ; irq13
                                                    ; 对应定时器
```

2 更新中断

```
                dc.l {$82000000+NonHandledInterrupt}      ; irq14
                dc.l {$82000000+NonHandledInterrupt}      ; irq15
                dc.l {$82000000+NonHandledInterrupt}      ; irq16
                dc.l {$82000000+NonHandledInterrupt}      ; irq17
                dc.l {$82000000+NonHandledInterrupt}      ; irq18
                dc.l {$82000000+NonHandledInterrupt}      ; irq19
                dc.l {$82000000+NonHandledInterrupt}      ; irq20
                dc.l {$82000000+NonHandledInterrupt}      ; irq21
                dc.l {$82000000+NonHandledInterrupt}      ; irq22
                dc.l {$82000000+NonHandledInterrupt}      ; irq23
                dc.l {$82000000+NonHandledInterrupt}      ; irq24
                dc.l {$82000000+NonHandledInterrupt}      ; irq25
                dc.l {$82000000+NonHandledInterrupt}      ; irq26
                dc.l {$82000000+NonHandledInterrupt}      ; irq27
                dc.l {$82000000+NonHandledInterrupt}      ; irq28
                dc.l {$82000000+NonHandledInterrupt}      ; irq29
        end
```

STM8 与汇编语言（ 9）－－ UART 应用之一

STM8 单片机的 UART 功能很强， 即可以作为普通的 UART 来使用， 也支持 LIN（局部互连网）。这里给出的例子是作为普通的 UART 来使用的。

在使用 UART 时，主要面向的寄存器有：控制寄存器 CR1、CR2 和 CR3，数据寄存器 DR，状态寄存器 SR，还有波特率寄存器 BRR1 和 BRR2。

这里特别要指出的是波特率寄存器是一个比较怪的设计，也说不清为啥这么设计，反正用起来相当别扭。 例如，当主时钟为 2MHZ 时，如果要求波特率为 9600，则分频系数 DIV=2000000/9600=208，变成 16 进制数就是 00D0。按照常规理解，那就是 00 送波特率分频寄存器的高 8 位， D0 送波特率分频寄存器的低 8 位。但在 STM8 单片机中却不是这样， BRR1 保存的是分频系数的第 11 位到第 4 位，BRR2 保存的是分频系数的第 15 位到第 12 位，和第 3 位到第 0 位。那么对于这个例子来说， BRR1=0D,BRR2=00。

另外一点要注意，必须先写 BRR2，然后再写 BRR1。
同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。
编译通过后，下载到开发板，运行程序。在 PC 机上用串口调试工具软件，将波特率设置在 9600，可以看到接收的字符为 ABABAB 。
另外做实验时，如果将 ST 的三合一板与 PC 机相连时要注意， DB9 的 2 和 3 引脚要交换，即三合一板子的 DB9 的第 3 脚要连到 PC 机的 DB9 的第 2 脚，而三合一板子的 DB9 的第 2 脚则要连到 PC 机的 DB9 的第 3 脚，当然第 5 脚（GND）就直连了。

```
stm8/

        #include "mapping.inc"

        #include "STM8S207C_S.INC"

; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end

                segment'rom'            ; 下面开始定义一个段，该段位于 ROM 中
main.l                                  ; 定义复位后的第一条指令的标号 （即入口地址）
;
; 首先要初始化堆栈指针
    LDW     X,#stack_end
        LDW     SP,X

        CALL    UART3_Init              ; 初始化串口 3
MAIN_LOOP.L
                LD      A,#$41
                CALL    UART3_SendChar
                LD      A,#$42
                CALL    UART3_SendChar
        JRA     MAIN_LOOP               ; 进入无限循环

UART3_Init.l                            ; 串口初始化子程序
        LD      A,#0                    ; 禁止 UART 发送和接收
        LD      LINUART_CR2,A
```

```
        LD      A,#0
        LD      LINUART_CR1,A           ; b5 = 0,允许 UART
                                        ; b2 = 0,禁止校验

        LD      A,#0                    ; b5,b4 = 00,1 个停止位
        LD      LINUART_CR3,A

; 设置波特率，必须注意以下几点：
;   (1) 必须先写 BRR2
;   (2) BRR1 存放的是分频系数的第 11 位到第 4 位,
;   (3) BRR2 存放的是分频系数的第 15 位到第 12 位，和第 3 位到第 0 位
; 例如对于波特率位 9600 时，分频系数 =2000000/9600=208
; 对应的十六进制数为 00D0， BBR1=0D,BBR2=00
        LD      A,#$00
        LD      LINUART_BRR2,A
        LD      A,#$0D
        LD      LINUART_BRR1,A          ; 实际的波特率分频系数为 00D0(208)
                                        ; 对应的波特率为 2000000/208=9600
        LD      A,#$0C                  ; b3 = 1,允许发送
                                        ; b2 = 1,允许接收
        LD      LINUART_CR2,A
        RET
;
UART3_SendChar.l                        ; 发送字符的子程序
        PUSH    A                       ; 将要发送的字符保存到堆栈中
SENDCHAR_1.L
        LD      A,LINUART_SR            ; 读取当前状态寄存器的值
        AND     A,#$80                  ; 若发送寄存器不空，则等待
        JREQ    SENDCHAR_1
        POP     A                       ; 从堆栈中恢复要发送的字符
        LD      LINUART_DR,A            ; 将要发送的字符送到数据寄存器
        RET


  interrupt NonHandledInterrupt
NonHandledInterrupt.l
        iret


; 下面定义中断向量表
        segment'vectit'
        dc.l {$82000000+main}                   ; reset
        dc.l {$82000000+NonHandledInterrupt}    ; trap
        dc.l {$82000000+NonHandledInterrupt}    ; irq0
```

```
dc.l {$82000000+NonHandledInterrupt}      ; irq1
dc.l {$82000000+NonHandledInterrupt}      ; irq2
dc.l {$82000000+NonHandledInterrupt}      ; irq3
dc.l {$82000000+NonHandledInterrupt}      ; irq4
dc.l {$82000000+NonHandledInterrupt}      ; irq5
dc.l {$82000000+NonHandledInterrupt}      ; irq6
dc.l {$82000000+NonHandledInterrupt}      ; irq7
dc.l {$82000000+NonHandledInterrupt}      ; irq8
dc.l {$82000000+NonHandledInterrupt}      ; irq9
dc.l {$82000000+NonHandledInterrupt}      ; irq10
dc.l {$82000000+NonHandledInterrupt}      ; irq11
dc.l {$82000000+NonHandledInterrupt}      ; irq12
dc.l {$82000000+NonHandledInterrupt}      ; irq13
dc.l {$82000000+NonHandledInterrupt}      ; irq14
dc.l {$82000000+NonHandledInterrupt}      ; irq15
dc.l {$82000000+NonHandledInterrupt}      ; irq16
dc.l {$82000000+NonHandledInterrupt}      ; irq17
dc.l {$82000000+NonHandledInterrupt}      ; irq18
dc.l {$82000000+NonHandledInterrupt}      ; irq19
dc.l {$82000000+NonHandledInterrupt}      ; irq20
dc.l {$82000000+NonHandledInterrupt}      ; irq21
dc.l {$82000000+NonHandledInterrupt}      ; irq22
dc.l {$82000000+NonHandledInterrupt}      ; irq23
dc.l {$82000000+NonHandledInterrupt}      ; irq24
dc.l {$82000000+NonHandledInterrupt}      ; irq25
dc.l {$82000000+NonHandledInterrupt}      ; irq26
dc.l {$82000000+NonHandledInterrupt}      ; irq27
dc.l {$82000000+NonHandledInterrupt}      ; irq28
dc.l {$82000000+NonHandledInterrupt}      ; irq29


    end
```

STM8 与汇编语言（ 10）－－ UART 应用之二

下面这个实验程序比较简单，它是在上篇基础上，增加了查询方式从 UART 接收一个字符的子程序。 主循环中， 等待接收一个字符， 然后将接收到的字符再发送出去。如果与 WINDOWS 的超级终端相连，则键盘上按什么按键，则显示对应的字符。

同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。

编译通过后，下载到开发板，运行程序。在 PC 机上运行超级终端， 波特率为 9600，然后从键盘上输入按键，屏幕上就显示对应的字符。

```
stm8/

        #include "mapping.inc"

        #include "STM8S207C_S.INC"

; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end



                segment'rom'                ; 下面开始定义一个段，该段位于 ROM 中
main.l                                      ; 定义复位后的第一条指令的标号 （即入口地址）
;
; 首先要初始化堆栈指针
        LDW     X,#stack_end
                LDW     SP,X

                CALL    UART3_Init          ; 初始化串口 3
MAIN_LOOP.L
                CALL    UART3_RecvChar
                CALL    UART3_SendChar
                JRA     MAIN_LOOP           ; 进入无限循环


UART3_Init.l                                ; 串口初始化子程序
                LD      A,#0                ; 禁止 UART 发送和接收
                LD      LINUART_CR2,A

                LD      A,#0
                LD      LINUART_CR1,A       ; b5 = 0,允许 UART
                                            ; b2 = 0,禁止校验

                LD      A,#0                ; b5,b4 = 00,1 个停止位
                LD      LINUART_CR3,A

; 设置波特率，必须注意以下几点：
;   (1) 必须先写 BRR2
;   (2) BRR1 存放的是分频系数的第 11 位到第 4 位,
```

```
;      (3) BRR2 存放的是分频系数的第    15 位到第  12 位，和第  3 位到第  0 位
; 例如对于波特率位    9600 时，分频系数  =2000000/9600=208
; 对应的十六进制数为    00D0，  BBR1=0D,BBR2=00
                LD        A,#$00
                LD        LINUART_BRR2,A
                LD        A,#$0D
                LD        LINUART_BRR1,A          ; 实际的波特率分频系数为
00D0(208)
                                                  ; 对应的波特率为
2000000/208=9600
                LD        A,#$0C                  ; b3 = 1,允许发送
                                                  ; b2 = 1,允许接收
                LD        LINUART_CR2,A
                RET
;
UART3_SendChar.l                                  ; 发送字符的子程序
                PUSH     A                        ; 将要发送的字符保存到堆栈中
SENDCHAR_1.L
                LD        A,LINUART_SR            ; 读取当前状态寄存器的值
                AND       A,#$80                  ; 若发送寄存器不空，则等待
                JREQ      SENDCHAR_1
                POP       A                       ; 从堆栈中恢复要发送的字符
                LD        LINUART_DR,A            ; 将要发送的字符送到数据寄存器
                RET

; 函数功能：从   UART3 接收一个字符
; 输入参数：无
; 输出参数：无
; 返  回  值：寄存器  A --  从串口读回的字符
; 备      注：无
UART3_RecvChar.l
RECVCHAR_1.L
                LD        A,LINUART_SR
                AND       A,#$20
                JREQ      RECVCHAR_1
                LD        A,LINUART_DR
                RET


  interrupt NonHandledInterrupt
NonHandledInterrupt.l
                iret
```

```
; 下面定义中断向量表
        segment'vectit'
        dc.l {$82000000+main}                   ; reset
        dc.l {$82000000+NonHandledInterrupt}    ; trap
        dc.l {$82000000+NonHandledInterrupt}    ; irq0
        dc.l {$82000000+NonHandledInterrupt}    ; irq1
        dc.l {$82000000+NonHandledInterrupt}    ; irq2
        dc.l {$82000000+NonHandledInterrupt}    ; irq3
        dc.l {$82000000+NonHandledInterrupt}    ; irq4
        dc.l {$82000000+NonHandledInterrupt}    ; irq5
        dc.l {$82000000+NonHandledInterrupt}    ; irq6
        dc.l {$82000000+NonHandledInterrupt}    ; irq7
        dc.l {$82000000+NonHandledInterrupt}    ; irq8
        dc.l {$82000000+NonHandledInterrupt}    ; irq9
        dc.l {$82000000+NonHandledInterrupt}    ; irq10
        dc.l {$82000000+NonHandledInterrupt}    ; irq11
        dc.l {$82000000+NonHandledInterrupt}    ; irq12
        dc.l {$82000000+NonHandledInterrupt}    ; irq13
        dc.l {$82000000+NonHandledInterrupt}    ; irq14
        dc.l {$82000000+NonHandledInterrupt}    ; irq15
        dc.l {$82000000+NonHandledInterrupt}    ; irq16
        dc.l {$82000000+NonHandledInterrupt}    ; irq17
        dc.l {$82000000+NonHandledInterrupt}    ; irq18
        dc.l {$82000000+NonHandledInterrupt}    ; irq19
        dc.l {$82000000+NonHandledInterrupt}    ; irq20
        dc.l {$82000000+NonHandledInterrupt}    ; irq21
        dc.l {$82000000+NonHandledInterrupt}    ; irq22
        dc.l {$82000000+NonHandledInterrupt}    ; irq23
        dc.l {$82000000+NonHandledInterrupt}    ; irq24
        dc.l {$82000000+NonHandledInterrupt}    ; irq25
        dc.l {$82000000+NonHandledInterrupt}    ; irq26
        dc.l {$82000000+NonHandledInterrupt}    ; irq27
        dc.l {$82000000+NonHandledInterrupt}    ; irq28
        dc.l {$82000000+NonHandledInterrupt}    ; irq29
```

STM8 与汇编语言（ 11）－－ UART 应用之三

下面这个实验程序是在上一个实验程序的基础上，将字符接收改成中断方式。 每当接收到一个字符，进入中断服务程序，在中断服务程序中，从 UART 的接收

数据寄存器中读出字符， 然后通过字符发送子程序发送出去。 如果与 WINDOWS 的超级终端相连，则键盘上按什么按键，则显示对应的字符。

同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。要注意的是，一定要将串口的接收中断服务程序的入口地址填写中断向量表中正确的位置。

编译通过后，下载到开发板，运行程序。在 PC 机上运行超级终端， 波特率为 9600，然后从键盘上输入按键，屏幕上就显示对应的字符。

```
stm8/

        #include "mapping.inc"

        #include "STM8S207C_S.INC"

; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end

; 下面开始定义一个段，该段位于 ROM 中
                segment'rom'
; 定义复位后的第一条指令的标号（即入口地址）
main.l
;
; 首先要初始化堆栈指针
            LDW     X,#stack_end
            LDW     SP,X


;
            CALL    UART3_Init              ; 初始化串口 3
            RIM                             ; 允许 CPU 全局中断
MAIN_LOOP.L
            JRA     MAIN_LOOP               ; 进入无限循环
;
; UART3 接收中断服务程序
  interrupt UART3_Recv_ISR
UART3_Recv_ISR.L
                PUSH    A
                LD      A,LINUART_DR            ; 读入接收到的字符
```

```
                CALL      UART3_SendChar          ; 将字符发送出去
                POP       A
               IRET

;
; 函数功能：初始化  UART3
; 输入参数：无
; 输出参数：无
; 返  回  值：无
; 备      注：寄存器  A --  被修改掉
UART3_Init.I                                ; 串口初始化子程序
        LD      A,#0                        ; 禁止 UART 发送和接收
        LD      LINUART_CR2,A

        LD      A,#0
        LD      LINUART_CR1,A           ; b5 = 0,允许 UART
                                        ; b2 = 0,禁止校验

        LD      A,#0                    ; b5,b4 = 00,1 个停止位
        LD      LINUART_CR3,A

; 设置波特率，必须注意以下几点：
;     (1)  必须先写  BRR2
;     (2) BRR1 存放的是分频系数的第   11 位到第  4 位,
;     (3) BRR2 存放的是分频系数的第   15 位到第  12 位，和第  3 位到第  0 位
; 例如对于波特率位   9600 时，分频系数  =2000000/9600=208
; 对应的十六进制数为   00D0，  BBR1=0D,BBR2=00
        LD      A,#$00
        LD      LINUART_BRR2,A
        LD      A,#$0D
        LD      LINUART_BRR1,A          ; 实际的波特率分频系数为   00D0(208)
                                        ; 对应的波特率为  2000000/208=9600
        LD      A,#$2C                  ; b3 = 1,允许发送
                                        ; b2 = 1,允许接收
                                        ; b5 = 1,允许产生接收中断
        LD      LINUART_CR2,A
        RET
;
; 函数功能：从  UART3 发送一个字符
; 输入参数：寄存器   A --  要发送的字符
; 输出参数：无
; 返  回  值：无
; 备      注：无
UART3_SendChar.I                        ; 发送字符的子程序
```

```
        PUSH    A                       ; 将要发送的字符保存到堆栈中
SENDCHAR_1.L
        LD      A,LINUART_SR            ; 读取当前状态寄存器的值
        AND     A,#$80                  ; 若发送寄存器不空，则等待
        JREQ    SENDCHAR_1
        POP     A                       ; 从堆栈中恢复要发送的字符
        LD      LINUART_DR,A            ; 将要发送的字符送到数据寄存器
        RET

  interrupt NonHandledInterrupt
NonHandledInterrupt.l
                iret


; 下面定义中断向量表
                segment'vectit'
                dc.l {$82000000+main}                                   ; reset
                dc.l {$82000000+NonHandledInterrupt}    ; trap
                dc.l {$82000000+NonHandledInterrupt}    ; irq0
                dc.l {$82000000+NonHandledInterrupt}    ; irq1
                dc.l {$82000000+NonHandledInterrupt}    ; irq2
                dc.l {$82000000+NonHandledInterrupt}    ; irq3
                dc.l {$82000000+NonHandledInterrupt}    ; irq4
                dc.l {$82000000+NonHandledInterrupt}    ; irq5
                dc.l {$82000000+NonHandledInterrupt}    ; irq6
                dc.l {$82000000+NonHandledInterrupt}    ; irq7
                dc.l {$82000000+NonHandledInterrupt}    ; irq8
                dc.l {$82000000+NonHandledInterrupt}    ; irq9
                dc.l {$82000000+NonHandledInterrupt}    ; irq10
                dc.l {$82000000+NonHandledInterrupt}    ; irq11
                dc.l {$82000000+NonHandledInterrupt}    ; irq12
                dc.l {$82000000+NonHandledInterrupt}    ; irq13
                dc.l {$82000000+NonHandledInterrupt}    ; irq14
                dc.l {$82000000+NonHandledInterrupt}    ; irq15
                dc.l {$82000000+NonHandledInterrupt}    ; irq16
                dc.l {$82000000+NonHandledInterrupt}    ; irq17
                dc.l {$82000000+NonHandledInterrupt}    ; irq18
                dc.l {$82000000+NonHandledInterrupt}    ; irq19
                dc.l {$82000000+NonHandledInterrupt}    ; irq20
                dc.l {$82000000+UART3_Recv_ISR}                 ; irq21
                                                        ; 对应串口 3 接收中断
        dc.l {$82000000+NonHandledInterrupt}    ; irq22
          dc.l {$82000000+NonHandledInterrupt}    ; irq23
          dc.l {$82000000+NonHandledInterrupt}    ; irq24
          dc.l {$82000000+NonHandledInterrupt}    ; irq25
```

```
  dc.l {$82000000+NonHandledInterrupt}       ; irq26
  dc.l {$82000000+NonHandledInterrupt}       ; irq27
  dc.l {$82000000+NonHandledInterrupt}       ; irq28
  dc.l {$82000000+NonHandledInterrupt}       ; irq29


end
```

STM8 与汇编语言（ 12）－－ EEPROM 应用

在单片机的应用系统中，经常会用到 EEPROM，用来保存一些掉电后仍然需要保存的数据。传统的方法是在单片机外再加一个 EEPROM 芯片，这种方法除了增加成本外， 也降低了可靠性。 现在，许多单片机芯片公司也都推出了集成有小容量 EEPROM 的单片机，降低了成本，提高了可靠性。

STM8 单片机芯片内部也集成有 EEPROM，容量从 640 字节到 2K 字节。最为关键的是，在 STM8 单片机中，访问 EEPROM 就向访问常规的内存一样，非常方便。 EEPROM 的地址空间与内存是统一编址的，地址从 004000H 开始，大小根据不同的芯片型号而定。

如果我们要读出 EEPROM 中的第一个单元的内容，则只要执行 LD A，$4000 这条指令，就可以将 EEPROM 中的第一个单元的内容读到累加器 A 中。

当需要将数据写入 EEPROM 中时，首先进行解锁操作，当解锁成功后，直接执行 LD $4000,A 这条指令， 就可以将累加器 A 中的值， 写入到 EEPROM 的第一个单元中。然后通过查询状态，判断写入操作是否成功。

下面的实验程序，就是先给 EEPROM 中的第一个单元 004000H 写入 34H，然后再读到累加器 A 中。

同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。

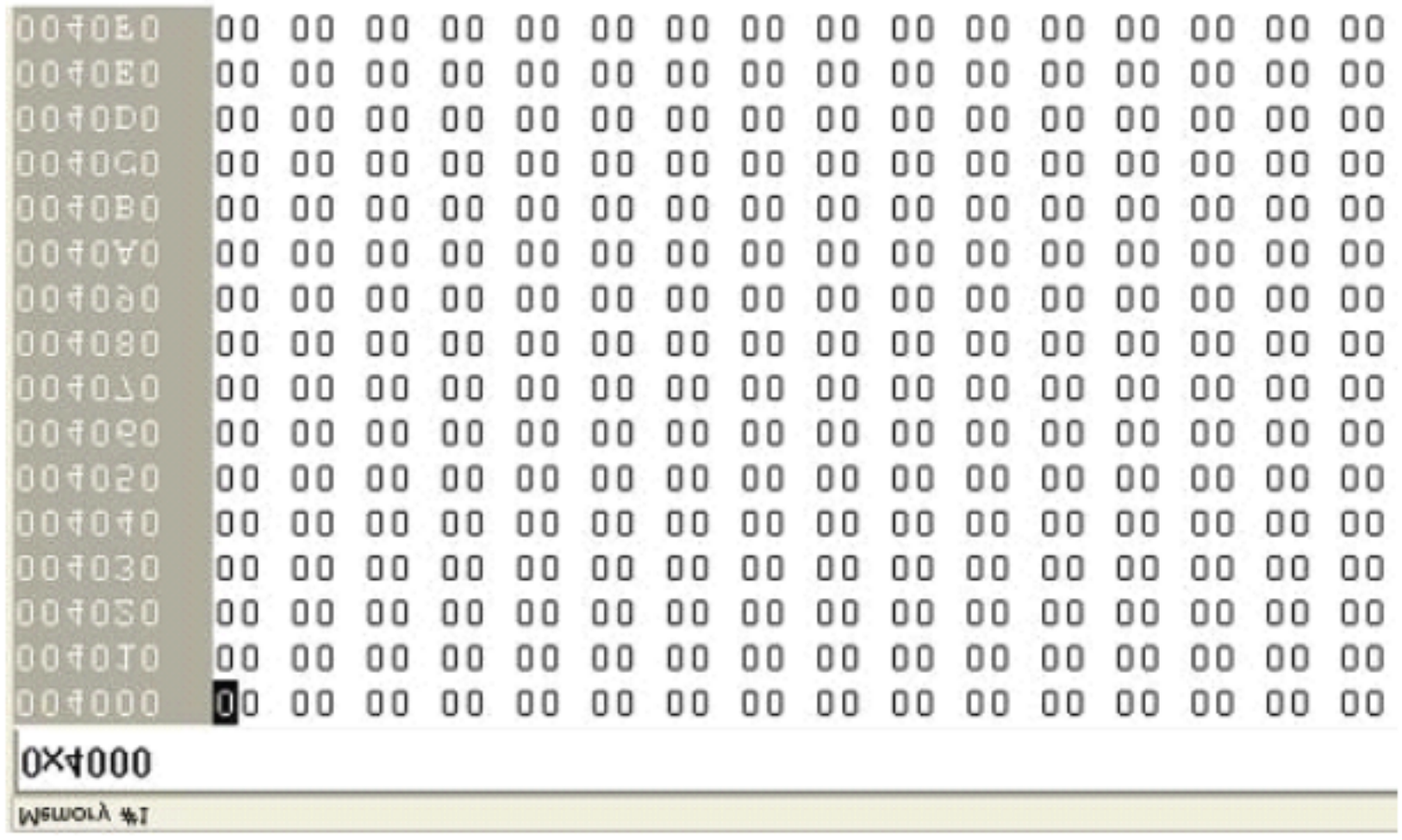
stm8/

```
        #include "mapping.inc"
        #include "STM8S207C_S.INC"


; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end
```

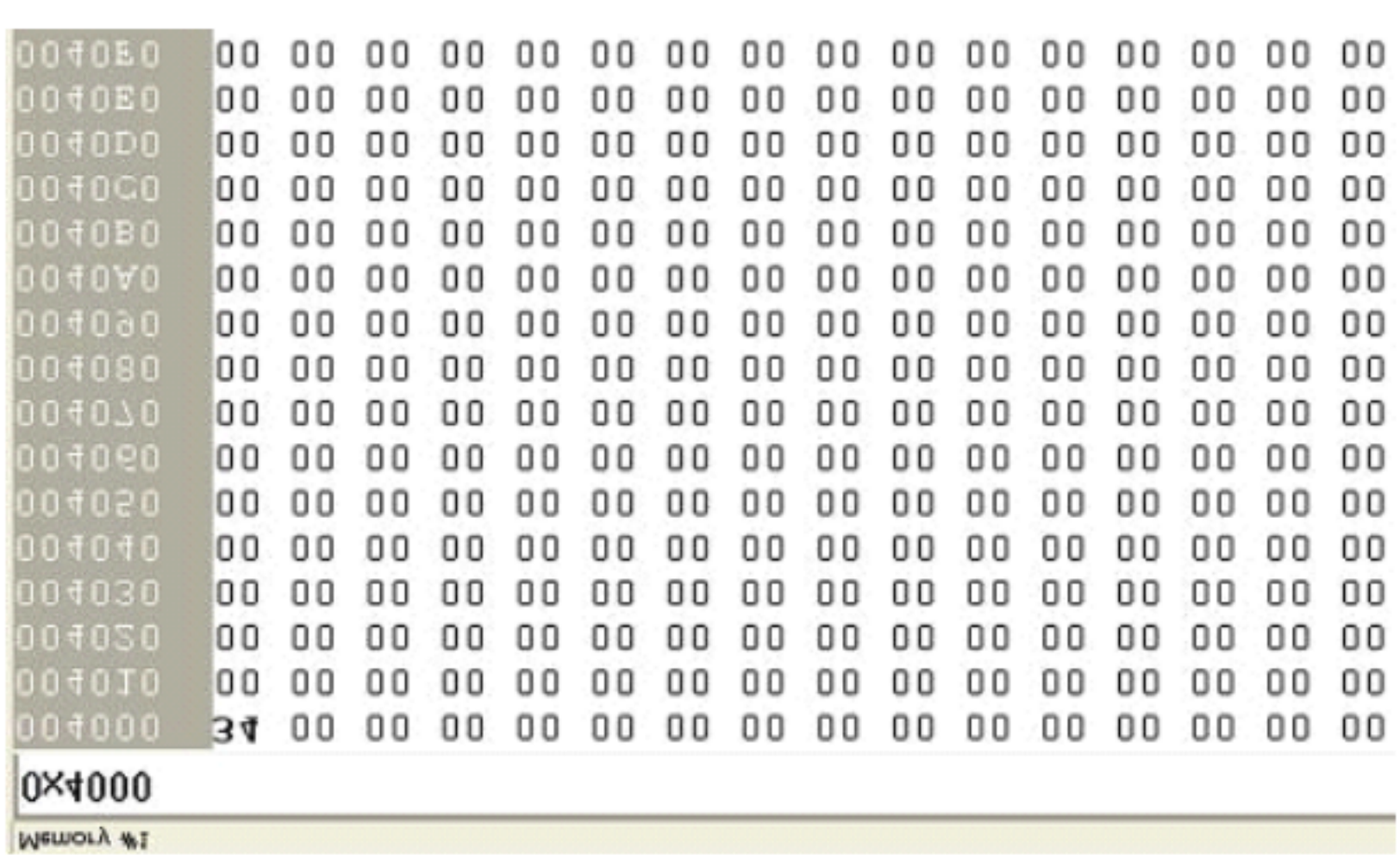
```
                segment'rom'          ; 下面开始定义一个段，该段位于 ROM 中
main.l                                ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
            LDW     X,#stack_end
            LDW     SP,X


; 对数据 EEPROM 进行解锁
WAIT_UNLOCK.L
            LD      A,#$AE
            LD      FLASH_DUKR,A          ; 写入第一个密钥
            LD      A,#$56
            LD      FLASH_DUKR,A          ; 写入第二个密钥

            LD      A,FLASH_IAPSR         ; 检查是否解锁成功
            AND     A,#$08
            JREQ    WAIT_UNLOCK           ; 若不成功，重新再来

            LD      A,#$34                ; 写入第一个字节
            LD      $4000,A


WAIT_WRITE_END.L
            LD      A,FLASH_IAPSR         ; 等待写操作结束
            AND     A,#$04
            JREQ    WAIT_WRITE_END

            LD      A,#$00                ; 先将累加器 A 清 0
            LD      A,$4000               ; 读出刚才写入的单元

MAIN_LOOP.L
            JRA     MAIN_LOOP             ; 进入无限循环
;

  interrupt NonHandledInterrupt
NonHandledInterrupt.l
            iret
```

```
; 下面定义中断向量表
        segment'vectit'
        dc.l {$82000000+main}                   ; reset
        dc.l {$82000000+NonHandledInterrupt}    ; trap
        dc.l {$82000000+NonHandledInterrupt}    ; irq0
        dc.l {$82000000+NonHandledInterrupt}    ; irq1
        dc.l {$82000000+NonHandledInterrupt}    ; irq2
        dc.l {$82000000+NonHandledInterrupt}    ; irq3
        dc.l {$82000000+NonHandledInterrupt}    ; irq4
        dc.l {$82000000+NonHandledInterrupt}    ; irq5
        dc.l {$82000000+NonHandledInterrupt}    ; irq6
        dc.l {$82000000+NonHandledInterrupt}    ; irq7
        dc.l {$82000000+NonHandledInterrupt}    ; irq8
        dc.l {$82000000+NonHandledInterrupt}    ; irq9
        dc.l {$82000000+NonHandledInterrupt}    ; irq10
        dc.l {$82000000+NonHandledInterrupt}    ; irq11
        dc.l {$82000000+NonHandledInterrupt}    ; irq12
        dc.l {$82000000+NonHandledInterrupt}    ; irq13
        dc.l {$82000000+NonHandledInterrupt}    ; irq14
        dc.l {$82000000+NonHandledInterrupt}    ; irq15
        dc.l {$82000000+NonHandledInterrupt}    ; irq16
        dc.l {$82000000+NonHandledInterrupt}    ; irq17
        dc.l {$82000000+NonHandledInterrupt}    ; irq18
        dc.l {$82000000+NonHandledInterrupt}    ; irq19
        dc.l {$82000000+NonHandledInterrupt}    ; irq20
        dc.l {$82000000+NonHandledInterrupt}    ; irq21
        dc.l {$82000000+NonHandledInterrupt}    ; irq22
        dc.l {$82000000+NonHandledInterrupt}    ; irq23
        dc.l {$82000000+NonHandledInterrupt}    ; irq24
        dc.l {$82000000+NonHandledInterrupt}    ; irq25
        dc.l {$82000000+NonHandledInterrupt}    ; irq26
        dc.l {$82000000+NonHandledInterrupt}    ; irq27
        dc.l {$82000000+NonHandledInterrupt}    ; irq28
        dc.l {$82000000+NonHandledInterrupt}    ; irq29
        end
```

编译通过后，下载到开发板。运行前先看一下 EEPROM 内存空间的值，如下图所示， 004000H 单元的值为 00H。



然后运行程序， 断点设置在 JRA MAIN_LOOP 这条指令上， 这时再看 EEPROM 中的内容如下图所示：



可见我们已经将 34H 这个值，写入到 EEPROM 空间中的第一个单元 004000H 中。

与其它公司单片机中的 EEPROM 相比，对 STM8 的 EEPROM 访问，确实要简单得多。

STM8 与汇编语言（ 13）——修改 CPU 的时钟

在有些单片机的应用系统中，并不需要 CPU 运行在多高的频率。在低频率下运行，芯片的功耗会大大下降。因此希望单片机能提供这个功能， STM8 单片机确实有这个功能，并且修改也非常方便。

下面的实验程序首先将 CPU 的运行时钟设置在 8MHZ，然后快速闪烁 LED 指示灯。接着，通过修改主时钟的分频系数和 CPU 时钟的分频系数，将 CPU 时钟频率设置在 500KHZ，然后再慢速闪烁 LED 指示灯。通过观察 LED 指示灯的闪烁频率，可以看到，同样的循环代码，由于 CPU 时钟频率的改变，闪烁频率和时间长短都发生了变化。

同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。

stm8/

```
        #include "mapping.inc"
        #include "STM8S207C_S.INC"

; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end

                segment'rom'            ; 下面开始定义一个段，该段位于 ROM 中
main.l                                  ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
        LDW     X,#stack_end
        LDW     SP,X

        LD      A,#08
        LD      PD_DDR,A                ; 将 PD3 设置成输出
        LD      A,#08
        LD      PD_CR1,A                ; 将 PD3 设置成推挽输出
        LD      A,#00
        LD      PD_CR2,A

        LD      A,#$E1
        LD      CLK_SWR,A               ;选择芯片内部的 16MHZ 的 RC 振荡器

                ;为主时钟
```

```
MAIN_LOOP.L
;下面设置  CPU 时钟分频器，使得   CPU 时钟 =主时钟
;通过发光二极管，可以看出，程序运行的速度确实明显提高了
        LD      A,#$08
        LD      CLK_CKDIVR,A        ；主时钟  = 16MHZ  / 2
                                    ; CPU 时钟  =  主时钟  = 8MHZ

        LD      A,#10               ; LED 高速闪 10 次
HIGH_SPEED.L
        PUSH    A                   ；保存寄存器
        LD      A,#08
        LD      PD_ODR,A            ；将 PD3 的输出设置成  1
        LD      A,#100
        CALL    DELAY_MS            ；延时 100MS

        LD      A,#00               ;
        LD      PD_ODR,A            ；将 PD3 的输出设置成  0
        LD      A,#100
        CALL    DELAY_MS            ；延时 100MS

        POP     A                   ；恢复寄存器
        DEC     A
        JRNE    HIGH_SPEED

;下面设置  CPU 时钟分频器，使得   CPU 时钟 =主时钟 /4
;通过发光二极管，可以看出，程序运行的速度确实明显下降了
        LD      A,#$1A              ;
        LD      CLK_CKDIVR,A        ；主时钟  = 16MHZ  / 8
                                                ; CPU 时钟  =  主时钟  /
4 = 500KHZ

        LD      A,#10               ; LED 低速闪 10 次
LOW_SPEED.L
        PUSH    A                   ；保存寄存器

        LD      A,#08
        LD      PD_ODR,A            ；将 PD3 的输出设置成  1
        LD      A,#100
        CALL    DELAY_MS            ；延时 100MS
```

```
LD      A,#00
LD      PD_ODR,A            ; 将 PD3 的输出设置成 0
LD      A,#100
CALL    DELAY_MS            ; 延时 100MS

POP     A                   ; 恢复寄存器
DEC     A
JRNE    LOW_SPEED

        JRA     MAIN_LOOP


; 函数功能：延时
; 输入参数：寄存器 A ——  要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
; 输出参数：无
; 返 回 值：无
; 备    注：无
DELAY_MS.L
        PUSH    A               ; 将入口参数保存到堆栈中
        LD      A,#250          ; 寄存器 A<-250,作为下面的循环数
DELAY_MS_1.L
        NOP                     ; 用空操作指令进行延时 4T
        NOP
        NOP
        NOP
        NOP
        DEC     A               ; 寄存器 A<-A-1，本条指令执行之间为 1T
        JRNE    DELAY_MS_1      ; 若不等于 0，则循环,
                                ; 本条指令执行时间为 2T（跳时）或 1T（不跳时）
POP     A                       ; 从堆栈中恢复入口参数
        DEC     A               ; 将要延时的 MS 数－1
        JRNE    DELAY_MS        ; 若不等于 0，则循环
        RET                     ; 函数返回


    interrupt NonHandledInterrupt
NonHandledInterrupt.l
        iret


; 下面定义中断向量表
        segment'vectit'
        dc.l {$82000000+main}                   ; reset
```

```
dc.l {$82000000+NonHandledInterrupt}      ; trap
dc.l {$82000000+NonHandledInterrupt}      ; irq0
dc.l {$82000000+NonHandledInterrupt}      ; irq1
dc.l {$82000000+NonHandledInterrupt}      ; irq2
dc.l {$82000000+NonHandledInterrupt}      ; irq3
dc.l {$82000000+NonHandledInterrupt}      ; irq4
dc.l {$82000000+NonHandledInterrupt}      ; irq5
dc.l {$82000000+NonHandledInterrupt}      ; irq6
dc.l {$82000000+NonHandledInterrupt}      ; irq7
dc.l {$82000000+NonHandledInterrupt}      ; irq8
dc.l {$82000000+NonHandledInterrupt}      ; irq9
dc.l {$82000000+NonHandledInterrupt}      ; irq10
dc.l {$82000000+NonHandledInterrupt}      ; irq11
dc.l {$82000000+NonHandledInterrupt}      ; irq12
dc.l {$82000000+NonHandledInterrupt}      ; irq13
dc.l {$82000000+NonHandledInterrupt}      ; irq14
dc.l {$82000000+NonHandledInterrupt}      ; irq15
dc.l {$82000000+NonHandledInterrupt}      ; irq16
dc.l {$82000000+NonHandledInterrupt}      ; irq17
dc.l {$82000000+NonHandledInterrupt}      ; irq18
dc.l {$82000000+NonHandledInterrupt}      ; irq19
dc.l {$82000000+NonHandledInterrupt}      ; irq20
dc.l {$82000000+NonHandledInterrupt}      ; irq21
dc.l {$82000000+NonHandledInterrupt}      ; irq22
dc.l {$82000000+NonHandledInterrupt}      ; irq23
dc.l {$82000000+NonHandledInterrupt}      ; irq24
dc.l {$82000000+NonHandledInterrupt}      ; irq25
dc.l {$82000000+NonHandledInterrupt}      ; irq26
dc.l {$82000000+NonHandledInterrupt}      ; irq27
dc.l {$82000000+NonHandledInterrupt}      ; irq28
dc.l {$82000000+NonHandledInterrupt}      ; irq29


end
```

STM8 与汇编语言（ 14）－－切换时钟源

STM8 单片机的时钟源， 即可以选内部的， 也可以选外部的， 在系统运行过程中，可以很方便地切换。
下面的实验程序首先将主时钟源切换到外部的晶体振荡器上，振荡频率为 8MHZ，然后，然后快速闪烁 LED 指示灯。接着，将主时钟源又切换到内部的振荡器上，振荡频率为 2MHZ，然后再慢速闪烁 LED 指示灯。通过观察 LED 指

示灯的闪烁频率，可以看到，同样的循环代码，由于主时钟源的改变的改变，　　闪烁频率和时间长短都发生了变化。
同样还是利用　ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。
stm8/

```
        #include "mapping.inc"
        #include "STM8S207C_S.INC"

; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end


                segment'rom'                ; 下面开始定义一个段，该段位于　ROM 中
main.l                                      ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
        LDW     X,#stack_end
        LDW     SP,X

        LD      A,#08
        LD      PD_DDR,A            ; 将 PD3 设置成输出
        LD      A,#08
        LD      PD_CR1,A            ; 将 PD3 设置成推挽输出
        LD      A,#00
        LD      PD_CR2,A            ;

        LD      A,#$01
        LD      CLK_ECKR,A          ; 允许外部高速振荡器工作
WAIT_HSE_READY.L
        LD      A,CLK_ECKR
        AND     A,#$02
        JREQ    WAIT_HSE_READY      ; 等待外部高速振荡器准备好

; 注意：经过上述切换后，　主时钟从 HSI/8（2MHZ）切换到 HSE（8MHZ）
;
MAIN_LOOP.L
;下面设置 CPU 时钟分频器，使得　CPU 时钟=主时钟
;通过发光二极管，可以看出，程序运行的速度确实明显提高了
        LD      A,#$02
        LD      CLK_SWCR,A          ; SWEN <- 1
```

```
        LD      A,#$B4
        LD      CLK_SWR,A          ; 选择芯片外部的高速振荡器为主时钟
WAIT_CLK_SWITCH_1.L
        LD      A,CLK_SWCR
        AND     A,#$08
        JREQ    WAIT_CLK_SWITCH_1  ; 等待切换成功

        LD      A,#$00             ; 清除切换标志
        LD      CLK_SWCR,A
```

```
        LD      A,#10              ; LED 高速闪 10 次
HIGH_SPEED.L
        PUSH    A                  ; 保存寄存器
        LD      A,#08
        LD      PD_ODR,A           ; 将 PD3 的输出设置成 1
        LD      A,#100
        CALL    DELAY_MS           ; 延时 100MS

        LD      A,#00
        LD      PD_ODR,A           ; 将 PD3 的输出设置成 1
        LD      A,#100
        CALL    DELAY_MS           ; 延时 100MS

        POP     A                  ; 恢复寄存器
        DEC     A
        JRNE    HIGH_SPEED
```

```
;下面设置 CPU 时钟分频器，使得 CPU 时钟=主时钟/4
;通过发光二极管，可以看出，程序运行的速度确实明显下降了
        LD      A,#$02
        LD      CLK_SWCR,A         ; SWEN <- 1

        LD      A,#$E1             ; 选择 HSI 为主时钟源
        LD      CLK_SWR,A
WAIT_CLK_SWITCH_2.L
        LD      A,CLK_SWCR
        AND     A,#$08
        JREQ    WAIT_CLK_SWITCH_2  ; 等待切换成功

        LD      A,#$00             ; 清除切换标志
        LD      CLK_SWCR,A
; 注意：经过上述切换后， 主时钟从 HSE（8MHZ）切换到 HSI/8（2MHZ）
```

```
        LD      A,#10               ; LED 低速闪 10 次
LOW_SPEED.L
        PUSH    A                   ; 保存寄存器

        LD      A,#08
        LD      PD_ODR,A            ; 将 PD3 的输出设置成 1
        LD      A,#100
        CALL    DELAY_MS            ; 延时 100MS

        LD      A,#00
        LD      PD_ODR,A            ; 将 PD3 的输出设置成 1
        LD      A,#100
        CALL    DELAY_MS            ; 延时 100MS

        POP     A                   ; 恢复寄存器
        DEC     A
        JRNE    LOW_SPEED

        JRA     MAIN_LOOP


; 函数功能：延时
; 输入参数：寄存器 A ——要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
; 输出参数：无
; 返 回 值：无
; 备    注：无
DELAY_MS.L
        PUSH    A                   ; 将入口参数保存到堆栈中
        LD      A,#250              ; 寄存器 A<-250,作为下面的循环数
DELAY_MS_1.L
        NOP                         ; 用空操作指令进行延时 4T
        NOP
        NOP
        NOP
        NOP
        DEC     A                   ; 寄存器 A<-A-1，本条指令执行之间为 1T
        JRNE    DELAY_MS_1          ; 若不等于 0，则循环,
                                    ; 本条指令执行时间为 2T（跳时）或 1T（不跳时）
        POP     A                   ; 从堆栈中恢复入口参数
        DEC     A                   ; 将要延时的 MS 数－1
        JRNE    DELAY_MS            ; 若不等于 0，则循环
        RET                         ; 函数返回
```

```
   interrupt NonHandledInterrupt
NonHandledInterrupt.l
            iret


;  下面定义中断向量表
            segment'vectit'
            dc.l {$82000000+main}                                    ; reset
            dc.l {$82000000+NonHandledInterrupt}       ; trap
            dc.l {$82000000+NonHandledInterrupt}       ; irq0
            dc.l {$82000000+NonHandledInterrupt}       ; irq1
            dc.l {$82000000+NonHandledInterrupt}       ; irq2
            dc.l {$82000000+NonHandledInterrupt}       ; irq3
            dc.l {$82000000+NonHandledInterrupt}       ; irq4
            dc.l {$82000000+NonHandledInterrupt}       ; irq5
            dc.l {$82000000+NonHandledInterrupt}       ; irq6
            dc.l {$82000000+NonHandledInterrupt}       ; irq7
            dc.l {$82000000+NonHandledInterrupt}       ; irq8
            dc.l {$82000000+NonHandledInterrupt}       ; irq9
            dc.l {$82000000+NonHandledInterrupt}       ; irq10
            dc.l {$82000000+NonHandledInterrupt}       ; irq11
            dc.l {$82000000+NonHandledInterrupt}       ; irq12
            dc.l {$82000000+NonHandledInterrupt}       ; irq13
            dc.l {$82000000+NonHandledInterrupt}       ; irq14
            dc.l {$82000000+NonHandledInterrupt}       ; irq15
            dc.l {$82000000+NonHandledInterrupt}       ; irq16
            dc.l {$82000000+NonHandledInterrupt}       ; irq17
            dc.l {$82000000+NonHandledInterrupt}       ; irq18
            dc.l {$82000000+NonHandledInterrupt}       ; irq19
            dc.l {$82000000+NonHandledInterrupt}       ; irq20
            dc.l {$82000000+NonHandledInterrupt}       ; irq21
            dc.l {$82000000+NonHandledInterrupt}       ; irq22
            dc.l {$82000000+NonHandledInterrupt}       ; irq23
            dc.l {$82000000+NonHandledInterrupt}       ; irq24
            dc.l {$82000000+NonHandledInterrupt}       ; irq25
            dc.l {$82000000+NonHandledInterrupt}       ; irq26
             dc.l {$82000000+NonHandledInterrupt}       ; irq27
             dc.l {$82000000+NonHandledInterrupt}       ; irq28
             dc.l {$82000000+NonHandledInterrupt}       ; irq29
           end
```

STM8 与汇编语言（ 15）——AD 转换

现在大部分的单片机也都具备了　A/D 转换器，有 8 位的，也有 10 位的，当然性能好的具备了 12 位的 A/D。在 STM8 单片机中，提供的是 10 位的 A/D，通道数随芯片不同而不同，少的有　4 个通道，多的则有　16 个通道。
下面的实验程序首先对　A/D 输入进行采样，然后将采样结果的高　8 位（丢弃最低的 2 位），作为延时参数去调用延时子程序，然后再去驱动　LED 控制信号。因此不同的采样值，决定了　LED 的闪烁频率。当旋转　ST 三合一开发板上的电位器时，可以看到　LED 的闪烁频率发生变化。
同样还是利用　ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。

```
stm8/


        #include "mapping.inc"
        #include "STM8S207C_S.INC"


; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end

        segment'rom'            ; 下面开始定义一个段，该段位于　ROM 中
main.l                          ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
        LDW     X,#stack_end
        LDW     SP,X

; 初始化 LED 对应的 IO 端口
        LD      A,#08
        LD      PD_DDR,A            ; 将 PD3 设置成输出
        LD      A,#08
        LD      PD_CR1,A            ; 将 PD3 设置成推挽输出
        LD      A,#00
        LD      PD_CR2,A            ;

; 初始化 A/D 模块
        LD      A,#$00
        LD      ADC_CR2,A           ; A/D 结果数据左对齐
        LD      A,#$00
```

```
        LD      ADC_CR1,A               ; ADC 时钟=主时钟/2=1MHZ
                                        ; ADC 转换模式 =单次
                                        ; 禁止 ADC 转换
        LD      A,#$03
        LD      ADC_CSR,A               ; 选择通道 3
        LD      A,#$20
        LD      ADC_TDRL,A
;
MAIN_LOOP.L
        LD      A,ADC_CR1
        OR      A,#$01
        LD      ADC_CR1,A               ; CR1 寄存器的最低位置 1, 使能 ADC 转换
        LD      A,#100
WAIT_ADC_ON.L
        DEC     A
        JRNE    WAIT_ADC_ON             ; 延时一段时间, 至少 7uS, 保证 ADC 模
块的上电完成
        LD      A,ADC_CR1
        OR      A,#$01
        LD      ADC_CR1,A               ; 再次将 CR1 寄存器的最低位置 1
                                        ; 使能 ADC 转换
WAIT_ADC_EOC.L
        LD      A,ADC_CSR
        AND     A,#$80
        JREQ    WAIT_ADC_EOC            ; 等待 ADC 结束

        LD      A,ADC_DRH               ; 读出 ADC 结果的高 8 位
        CALL    DELAY_MS                ; 延时一段时间

        LD      A,PD_ODR                ; 读回 PD 口的数据
        XOR     A,#$08                  ; 将 PD3 反相
        LD      PD_ODR,A                ; 送回


        JRA     MAIN_LOOP               ; 继续主循环

; 函数功能：延时
; 输入参数：寄存器 A ―― 要延时的毫秒数, 这里假设 CPU 的主频为 2MHZ
; 输出参数：无
; 返 回 值：无
; 备 注：无
DELAY_MS.L
        PUSH    A                       ; 将入口参数保存到堆栈中
        LD      A,#250                  ; 寄存器 A<-250, 作为下面的循环数
```

```
DELAY_MS_1.L
        NOP                         ; 用空操作指令进行延时    4T
        NOP
        NOP
        NOP
        NOP
        NOP
        DEC     A                   ; 寄存器 A<-A-1 ，本条指令执行之间为    1T
        JRNE    DELAY_MS_1          ; 若不等于 0，则循环,
                                    ; 本条指令执行时间为  2T（跳时）或 1T（不
跳时）
        POP     A                   ; 从堆栈中恢复入口参数
        DEC     A                   ; 将要延时的 MS 数－1
        JRNE    DELAY_MS            ; 若不等于 0，则循环
        RET                         ; 函数返回


  interrupt NonHandledInterrupt
NonHandledInterrupt.l
        iret


; 下面定义中断向量表
        segment'vectit'
        dc.l {$82000000+main}                   ; reset
        dc.l {$82000000+NonHandledInterrupt}    ; trap
        dc.l {$82000000+NonHandledInterrupt}    ; irq0
        dc.l {$82000000+NonHandledInterrupt}    ; irq1
        dc.l {$82000000+NonHandledInterrupt}    ; irq2
        dc.l {$82000000+NonHandledInterrupt}    ; irq3
        dc.l {$82000000+NonHandledInterrupt}    ; irq4
        dc.l {$82000000+NonHandledInterrupt}    ; irq5
        dc.l {$82000000+NonHandledInterrupt}    ; irq6
        dc.l {$82000000+NonHandledInterrupt}    ; irq7
        dc.l {$82000000+NonHandledInterrupt}    ; irq8
        dc.l {$82000000+NonHandledInterrupt}    ; irq9
        dc.l {$82000000+NonHandledInterrupt}    ; irq10
        dc.l {$82000000+NonHandledInterrupt}    ; irq11
        dc.l {$82000000+NonHandledInterrupt}    ; irq12
        dc.l {$82000000+NonHandledInterrupt}    ; irq13
        dc.l {$82000000+NonHandledInterrupt}    ; irq14
        dc.l {$82000000+NonHandledInterrupt}    ; irq15
        dc.l {$82000000+NonHandledInterrupt}    ; irq16
        dc.l {$82000000+NonHandledInterrupt}    ; irq17
        dc.l {$82000000+NonHandledInterrupt}    ; irq18
        dc.l {$82000000+NonHandledInterrupt}    ; irq19
```

```
dc.l {$82000000+NonHandledInterrupt}      ; irq20
dc.l {$82000000+NonHandledInterrupt}      ; irq21
dc.l {$82000000+NonHandledInterrupt}      ; irq22
dc.l {$82000000+NonHandledInterrupt}      ; irq23
dc.l {$82000000+NonHandledInterrupt}      ; irq24
dc.l {$82000000+NonHandledInterrupt}      ; irq25
dc.l {$82000000+NonHandledInterrupt}      ; irq26
dc.l {$82000000+NonHandledInterrupt}      ; irq27
dc.l {$82000000+NonHandledInterrupt}      ; irq28
dc.l {$82000000+NonHandledInterrupt}      ; irq29


        end
```

STM8 与汇编语言（ 16）－－ PWM

在单片机应用系统中，也常常会用到 PWM 信号输出，例如电机转速的控制。 现在很多高档的单片机也都集成了 PWM 功能模块，方便用户的应用。
对于 PWM 信号，主要涉及到两个概念，一个就是 PWM 信号的周期或频率， 另一个就是 PWM 信号的占空比。例如一个频率为 1KHZ，占空比为 30%，有效信号为 1 的 PWM 信号，在用示波器测量时，就是高电平的时间为 300uS，低电平的时间为 700uS 的周期波形。
在单片机中实现 PWM 信号的功能模块，实际上就是带比较器的计数器模块。 首先该计数器循环计数， 例如从 0 到 N，那么这个 N 就决定了 PWM 的周期，PWM 周期=（N+1）*计数器时钟的周期。在计数器模块中一定还有一个比较器，比较器有 2 个输入， 一个就是计数器的当前值， 另一个是可以设置的数， 这个数来自一个比较寄存器。当计数器的值小于比较寄存器的值时，输出为 1（可以设置为 0），当计数器的值大于或等于比较寄存器的值时，输出为 0（也可设置为 1，与前面对应）。
了解了这个基本原理后， 我们就可以使用 STM8 单片机中的 PWM 模块了。下面的实验程序首先将定时器 2 的通道 2 设置成 PWM 输出方式，然后通过设置自动装载寄存器 TIM2_CCR2，决定 PWM 信号的周期。在程序的主循环中，循环修改占空比，先是从 0 逐渐递增到 128，然后再从 128 递减到 0。
当把下面的程序在 ST 的三合一板上运行时， 可以看到发光二极管 LD1 逐渐变亮，然后又逐渐变暗，就这样循环往复。如果用示波器看，可以看到驱动 LD1 的信号波形的占空比从 0 变到 50%，然后又从 50%变到 0。
同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。
stm8/

```
        #include "mapping.inc"
        #include "STM8S207C_S.INC"


; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end


        segment'rom'            ; 下面开始定义一个段，该段位于 ROM 中
main.l                          ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
        LDW     X,#stack_end
        LDW     SP,X
;
        CALL    CLK_Init        ; 初始化时钟
        CALL    TIM_Init        ; 初始化定时器
;
; 下面的循环将占空比逐渐从 0 递增到 128
        LD      A,#$00
MAIN_LOOP.L
        PUSH    A               ; 保存当前占空比
        LD      A,#$00
        LD      TIM2_CCR2H,A
        POP     A
        LD      TIM2_CCR2L,A    ; 设置占空比
        PUSH    A               ; 保存当前占空比
        LD      A,#$5
        CALL    DELAY_MS        ; 延时 5MS
        POP     A               ; 恢复占空比
        INC     A               ; 当前占空比 +1
        CP      A,#128
        JRNE    MAIN_LOOP       ; 若不等于 128, 则循环
;
; 下面的循环将占空比逐渐从 128 递减到 0
MAIN_LOOP2.L
        PUSH    A               ; 保存当前占空比
        LD      A,#$00
        LD      TIM2_CCR2H,A
        POP     A
        LD      TIM2_CCR2L,A    ; 设置占空比
```

```
PUSH     A                    ; 保存当前占空比
LD       A,#$5
CALL     DELAY_MS             ; 延时 5MS
POP      A                    ; 恢复占空比
DEC      A                    ; 当前占空比 -1
CP       A,#$00
JRNE     MAIN_LOOP2           ; 若不等于 0，则循环

JRA      MAIN_LOOP            ; 重新开始大循环

; 函数功能：延时
; 输入参数：寄存器 A ——要延时的毫秒数，这里假设 CPU 的主频为 2MHZ
; 输出参数：无
; 返 回 值：无
; 备    注：无
DELAY_MS.L
        PUSH     A                    ; 将入口参数保存到堆栈中
        LD       A,#250               ; 寄存器 A<-250,作为下面的循环数
DELAY_MS_1.L
        NOP                           ; 用空操作指令进行延时 4T
        NOP
        NOP
        NOP
        NOP
        NOP
        DEC      A                    ; 寄存器 A<-A-1，本条指令执行之间为 1T
        JRNE     DELAY_MS_1           ; 若不等于 0，则循环,
                                      ; 本条指令执行时间为 2T（跳时）或 1T（不跳时）
        POP      A                    ; 从堆栈中恢复入口参数
        DEC      A                    ; 将要延时的 MS 数－1
        JRNE     DELAY_MS             ; 若不等于 0，则循环
        RET                           ; 函数返回


; 函数功能：初始化时钟
; 输入参数：无
; 输出参数：无
; 返 回 值：无
; 备    注：无
CLK_Init.L
        LD       A,#$E1
        LD       CLK_CMSR,A           ; HSI 作为主时钟源
        LD       A,CLK_CKDIVR
        AND      A,#$E7
```

```
        OR      A,#$10
        LD      CLK_CKDIVR,A        ; 10: fHSI= fHSI RC output/ 4
                                    ; fHSI = fHSI RC 输出/4 = 4MHZ
                                    ; 这个时钟也作为外设的时钟
        OR      A,#$01
        LD      CLK_CKDIVR,A        ; 001: fCPU="fMASTER/2". = 2MHZ
        RET
;
; 函数功能：初始化定时器 2 的通道 2，用于控制 LED 的亮度
; 输入参数：无
; 输出参数：无
; 返 回 值：无
; 备 注：无
TIM_Init.L
        LD      A,TIM2_CCMR2
        OR      A,#$70
        LD      TIM2_CCMR2,A  ; Output mode PWM2.                    */
                              ; 通道 2 被设置成比较输出方式
                              ; OC2M = 111,为 PWM 模式 2,
                              ; 向上计数时，若计数器小于比较值，为无效电平
                              ; 即当计数器在 0 到比较值时，输出为 1，否则为 0

        LD      A,TIM2_CCER1
        OR      A,#$30
        LD      TIM2_CCER1,A        ; CC polarity low,enable PWM output     */
                                    ; CC2P= 1，低电平为有效电平
                                    ; CC2E = 1，开启输出引脚

;初始化自动装载寄存器，决定 PWM 方波的频率,
Fpwm=4000000/256=15625HZ
        LD      A,#$00
        LD      TIM2_ARRH,A
        LD      A,#$FF
        LD      TIM2_ARRL,A

;初始化比较寄存器，决定 PWM 方波的占空比
        LD      A,#$00
        LD      TIM2_CCR2H,A
        LD      A,#$00
        LD      TIM2_CCR2L,A

;初始化时钟分频器为 1，即计数器的时钟频率为 Fmaster=4MHZ
        LD      A,#$00
        LD      TIM2_PSCR,A
```

```
;启动计数
        LD      A,TIM2_CR1
        OR      A,#$01
        LD      TIM2_CR1,A
        RET

  interrupt NonHandledInterrupt
NonHandledInterrupt.l
        iret


;  下面定义中断向量表
        segment'vectit'
        dc.l {$82000000+main}                   ; reset
        dc.l {$82000000+NonHandledInterrupt}    ; trap
        dc.l {$82000000+NonHandledInterrupt}    ; irq0
        dc.l {$82000000+NonHandledInterrupt}    ; irq1
        dc.l {$82000000+NonHandledInterrupt}    ; irq2
        dc.l {$82000000+NonHandledInterrupt}    ; irq3
        dc.l {$82000000+NonHandledInterrupt}    ; irq4
        dc.l {$82000000+NonHandledInterrupt}    ; irq5
        dc.l {$82000000+NonHandledInterrupt}    ; irq6
        dc.l {$82000000+NonHandledInterrupt}    ; irq7
        dc.l {$82000000+NonHandledInterrupt}    ; irq8
        dc.l {$82000000+NonHandledInterrupt}    ; irq9
        dc.l {$82000000+NonHandledInterrupt}    ; irq10
        dc.l {$82000000+NonHandledInterrupt}    ; irq11
        dc.l {$82000000+NonHandledInterrupt}    ; irq12
        dc.l {$82000000+NonHandledInterrupt}    ; irq13
        dc.l {$82000000+NonHandledInterrupt}    ; irq14
        dc.l {$82000000+NonHandledInterrupt}    ; irq15
        dc.l {$82000000+NonHandledInterrupt}    ; irq16
        dc.l {$82000000+NonHandledInterrupt}    ; irq17
        dc.l {$82000000+NonHandledInterrupt}    ; irq18
        dc.l {$82000000+NonHandledInterrupt}    ; irq19
        dc.l {$82000000+NonHandledInterrupt}    ; irq20
        dc.l {$82000000+NonHandledInterrupt}    ; irq21
        dc.l {$82000000+NonHandledInterrupt}    ; irq22
        dc.l {$82000000+NonHandledInterrupt}    ; irq23
        dc.l {$82000000+NonHandledInterrupt}    ; irq24
        dc.l {$82000000+NonHandledInterrupt}    ; irq25
        dc.l {$82000000+NonHandledInterrupt}    ; irq26
```

```
  dc.l {$82000000+NonHandledInterrupt}       ; irq27
  dc.l {$82000000+NonHandledInterrupt}       ; irq28
  dc.l {$82000000+NonHandledInterrupt}       ; irq29


end
```

STM8 与汇编语言（ 17）——蜂鸣器

蜂鸣器是现在单片机应用系统中很常见的，常用于实现报警功能。为此 STM8 特别集成了蜂鸣器模块，应用起来非常方便。

在应用蜂鸣器模块时， 首先要打开片内的低速 RC 振荡器（应该也能使用外部的高速时钟，不过本人没实验过） ，其频率为 128KHZ。然后通过设置蜂鸣器控制寄存器 BEEP_CSR 中的 BEEPDIV[4:0] 来获取 8KHZ 的时钟，再通过 BEEPSEL 最终产生 1KHZ 或 2KHZ 或 4KHZ 的蜂鸣器时钟，最后使能该寄存器中的 BEEPEN 位，产生蜂鸣器的输出。

下面的实验程序首先初始化低速振荡器，然后启动蜂鸣器，再延时 2.5 秒，然后关闭蜂鸣器。

同样还是利用 ST 的开发工具，生成一个汇编程序的框架，然后修改其中的 main.asm，修改后的代码如下。

stm8/

```
        #include "mapping.inc"
        #include "STM8S207C_S.INC"


; 定义堆栈空间的起始位置和结束位置
stack_start.w   EQU     $stack_segment_start
stack_end.w     EQU     $stack_segment_end


                segment'rom'            ; 下面开始定义一个段，该段位于 ROM 中
main.l                                  ; 定义复位后的第一条指令的标号（即入口地址）
;
; 首先要初始化堆栈指针
        LDW     X,#stack_end
        LDW     SP,X

        LD      A,CLK_ICKR
```

```
        OR      A,#$08
        LD      CLK_ICKR,A          ; 打开芯片内部的低速振荡器    LSI
WAIT_LSI_READY.L
        LD      A,CLK_ICKR
        AND     A,#$10
        JREQ    WAIT_LSI_READY      ; 等待振荡器稳定

        LD      A,#$2e              ; BEEPDIV[1:0]  = 00
                                    ; BEEPDIV[4:0]  = 0e
                                    ; BEEPEN        = 1
; 输出频率   = Fls / ( 8 * (BEEPDIV  + 2) )= 128K / (8 * 16) = 1K
        LD      BEEP_CSR,A          ; 打开蜂鸣器

        LD      A,#10               ; 延时 250MS*10
DELAY_1.L
        PUSH    A
        LD      A,#250              ; 延时 250MS
        CALL    DELAY_MS
        POP     A
        DEC     A
        JRNE    DELAY_1

        LD      A,#$1E              ; 关闭蜂鸣器
        LD      BEEP_CSR,A

MAIN_LOOP.L
        JRA     MAIN_LOOP


; 函数功能：延时
; 输入参数：寄存器  A  －－  要延时的毫秒数，这里假设   CPU 的主频为  2MHZ
; 输出参数：无
; 返  回  值：无
; 备      注：无
DELAY_MS.L
        PUSH    A                   ; 将入口参数保存到堆栈中
        LD      A,#250              ; 寄存器 A<-250,作为下面的循环数
DELAY_MS_1.L
        NOP                         ; 用空操作指令进行延时   4T
        NOP
        NOP
        NOP
        NOP
        NOP
        DEC     A                   ; 寄存器 A<-A-1 ， 本条指令执行之间为   1T
```

```
        JRNE    DELAY_MS_1          ; 若不等于 0，则循环,
                                    ; 本条指令执行时间为  2T（跳时）或 1T（不跳时）
        POP     A                   ; 从堆栈中恢复入口参数
        DEC     A                   ; 將要延時的 MS 数－1
        JRNE    DELAY_MS            ; 若不等于 0，则循环
        RET                         ; 函数返回
```

```
  interrupt NonHandledInterrupt
NonHandledInterrupt.l
                iret


; 下面定义中断向量表
        segment'vectit'
        dc.l {$82000000+main}                           ; reset
        dc.l {$82000000+NonHandledInterrupt}    ; trap
        dc.l {$82000000+NonHandledInterrupt}    ; irq0
        dc.l {$82000000+NonHandledInterrupt}    ; irq1
        dc.l {$82000000+NonHandledInterrupt}    ; irq2
        dc.l {$82000000+NonHandledInterrupt}    ; irq3
        dc.l {$82000000+NonHandledInterrupt}    ; irq4
        dc.l {$82000000+NonHandledInterrupt}    ; irq5
        dc.l {$82000000+NonHandledInterrupt}    ; irq6
        dc.l {$82000000+NonHandledInterrupt}    ; irq7
        dc.l {$82000000+NonHandledInterrupt}    ; irq8
        dc.l {$82000000+NonHandledInterrupt}    ; irq9
        dc.l {$82000000+NonHandledInterrupt}    ; irq10
        dc.l {$82000000+NonHandledInterrupt}    ; irq11
        dc.l {$82000000+NonHandledInterrupt}    ; irq12
        dc.l {$82000000+NonHandledInterrupt}    ; irq13
        dc.l {$82000000+NonHandledInterrupt}    ; irq14
        dc.l {$82000000+NonHandledInterrupt}    ; irq15
        dc.l {$82000000+NonHandledInterrupt}    ; irq16
        dc.l {$82000000+NonHandledInterrupt}    ; irq17
        dc.l {$82000000+NonHandledInterrupt}    ; irq18
        dc.l {$82000000+NonHandledInterrupt}    ; irq19
        dc.l {$82000000+NonHandledInterrupt}    ; irq20
        dc.l {$82000000+NonHandledInterrupt}    ; irq21
        dc.l {$82000000+NonHandledInterrupt}    ; irq22
        dc.l {$82000000+NonHandledInterrupt}    ; irq23
        dc.l {$82000000+NonHandledInterrupt}    ; irq24
        dc.l {$82000000+NonHandledInterrupt}    ; irq25
        dc.l {$82000000+NonHandledInterrupt}    ; irq26
```

```
  dc.l {$82000000+NonHandledInterrupt}      ; irq27
  dc.l {$82000000+NonHandledInterrupt}      ; irq28
  dc.l {$82000000+NonHandledInterrupt}      ; irq29


End
```