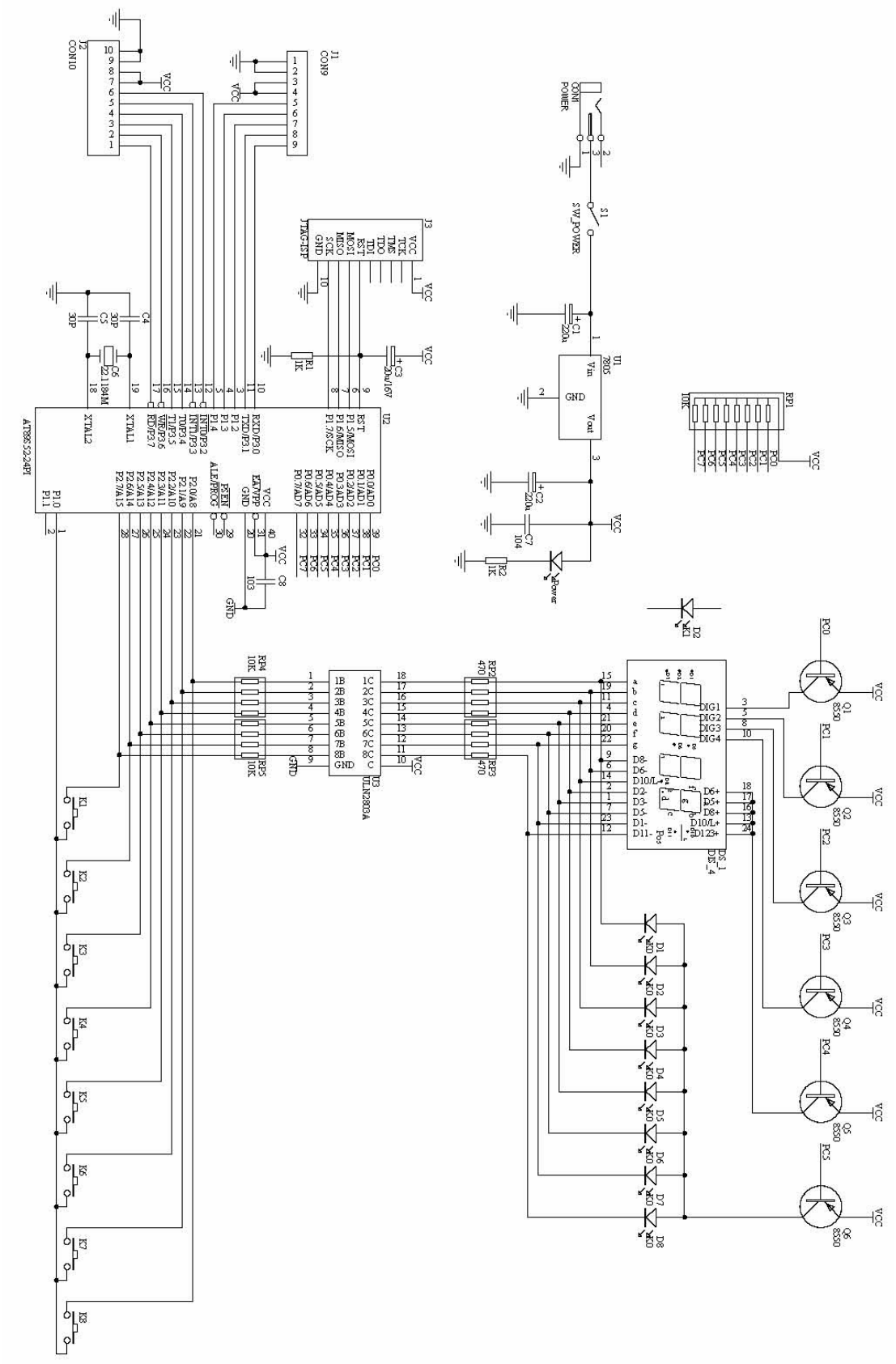
培训一：建立通用处理程序

—DIY 电子钟实例

研发中心 叶志伟 编写

一）原理图：



二）设计方案

1） K1 为设置开关键，用于开始进行时间设置和退出。（Setup）

2） K2 为向上减小键。（Up）

3） K3 为向下增大键。（Dowm）

4） K4 为确认键。（Return）

5） K1-K4 按下时，LED 的 D1-D4 点亮，松开时熄灭。

6） 其它 K5-K8 演示按键处理：

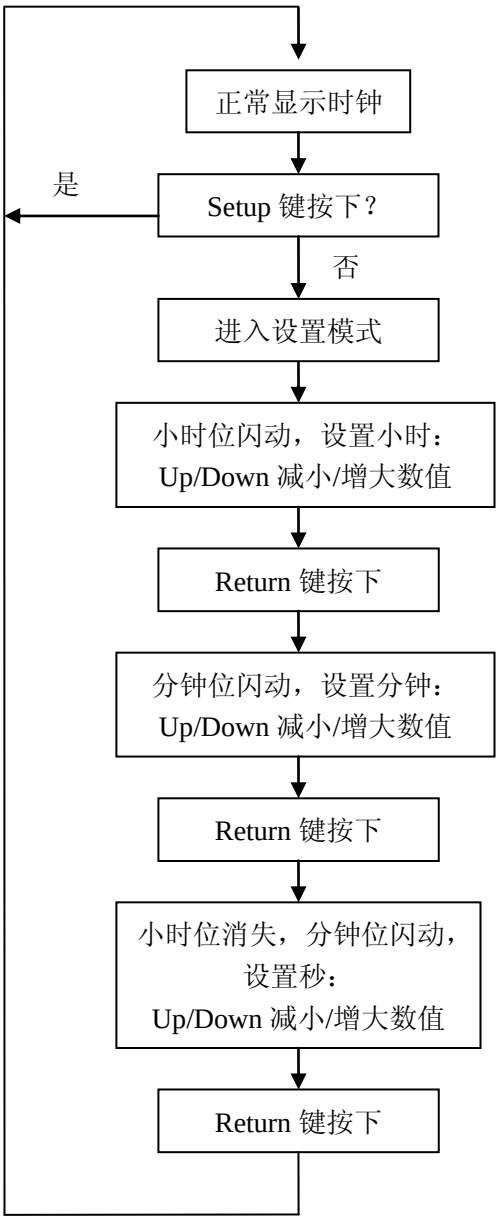
K5：快速连续两次按键时翻转 D5 状态。

K6：长按键翻转 D6 状态。

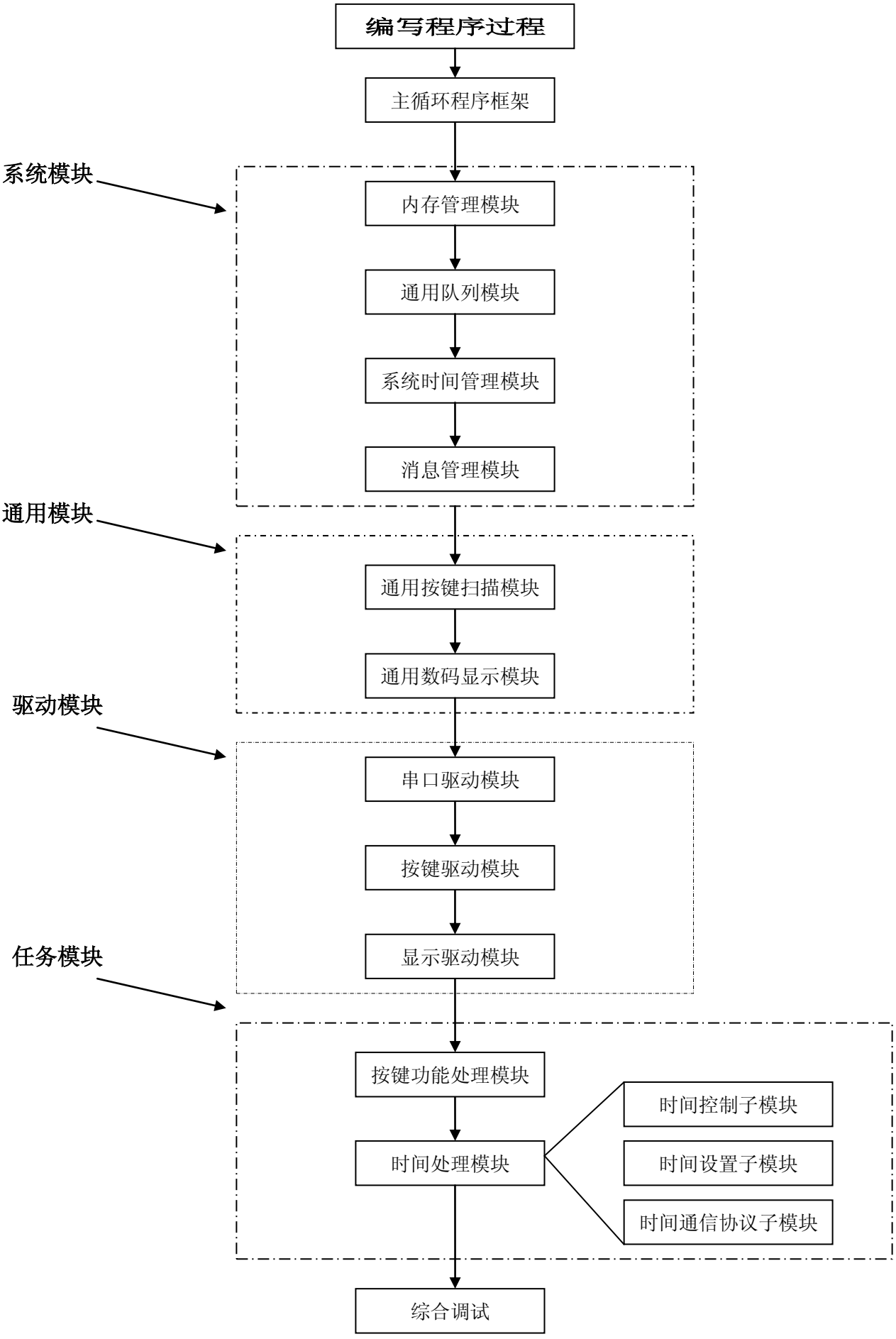
K7：在按下时翻转 D7 状态。

K8：在松开时翻转 D8 状态。

工作流程



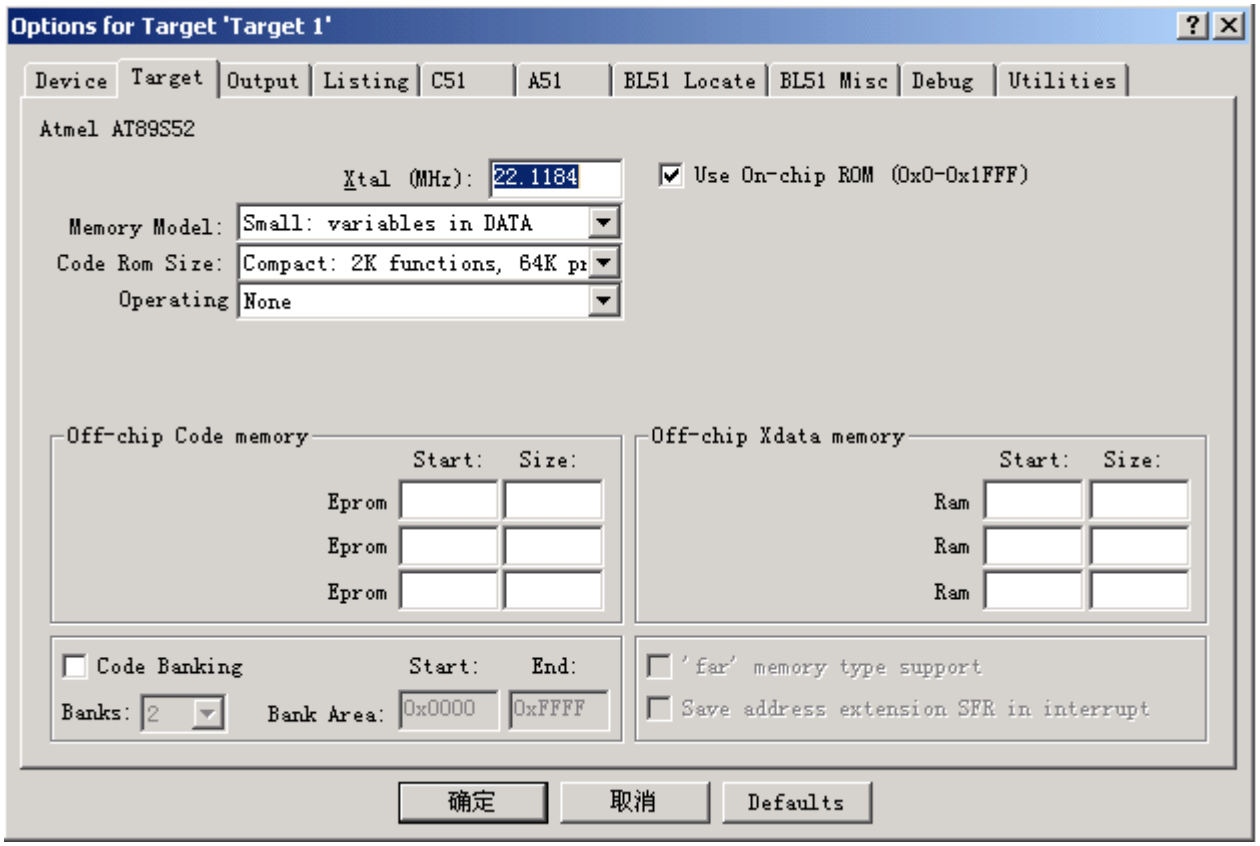
三）设计过程


编写程序过程
主循环程序框架
系统模块
内存管理模块
通用队列模块
系统时间管理模块
消息管理模块
通用模块
通用按键扫描模块
通用数码显示模块
驱动模块
串口驱动模块
按键驱动模块
显示驱动模块
任务模块
按键功能处理模块
时间处理模块
时间控制子模块
时间设置子模块
时间通信协议子模块
综合调试

四）设计过程说明：

1） 第一步：在 keil c51 编译环境中建立项目文件，配置和选项如下表：

	设置内容	备注	Keil 标签
MCU 型号	ATMEL 公司的 AT89S52		Device
工作频率	22.1184MHz		Xtal (MHz)
内存模式	Small	绝大部分程序都工作在这个模式下	Memory Model
程序代码模式	Compact		Code Rom Size
操作系统	None		Operating
使用内部程序区	是		Use On-chip ROM (0x0-0x1fff)
代码优化	6 级	大部分程序的优化等级	Code Optimization



2） 建立主循环框架函数 main.c 并添加到项目中。

代码如下：

```
//-------------------------------------------------------------
//主程序
//-------------------------------------------------------------
#include <reg52.h>

void main(void)
{

    EA = 1;

    while(1)
    {
    }
}
```

3）编写通用内存管理模块并添加到项目中。

该内存管理模块用来代替标准库中的 malloc 函数，用来支持通用函数和模块初始化加载时的内存申请，是标准化和规范化设计中必不可少的模块。

该内存管理，只分配内存不释放内存，且只在模块初始化函数中调用，不能在 while(1)主循环中被调用。相当于在模块初始化时才确定的全局变量（可以这么想象）。

C 代码 Memory.c 并添加到项目中。

```
//-------------------------------------------------------------------------
//内存管理
//-------------------------------------------------------------------------
#define _MemoryH

#include <reg52.h>
#include "Memory.h"

//-----------------------------------------------------------
//使用的变量
struct InMemory{
     unsigned char idata *pREnd;
};
struct InMemory sInMemory;



//-------------------------------------------------------------------------
//内存初始化
//内存初试指针总是指向 RAM 最后
//-------------------------------------------------------------------------
void Memory_Init(unsigned char idata *pRamEnd)
{
     sInMemory.pREnd = pRamEnd;
}

//-------------------------------------------------------------------------
//析构
//-------------------------------------------------------------------------
void Memory_Destory(void)
{
}

//-------------------------------------------------------------------------
//内存分配
//返回分配后的起始指针
//-------------------------------------------------------------------------
unsigned char * Memory_Malloc( unsigned char mSize )
{
     if( mSize ){
          sInMemory.pREnd -= mSize;
          return sInMemory.pREnd+1;
     }
```

```
    else {
        return 0;
    }
}

//---------------------------------------------------------------------------
//用指定数据清除内存
//---------------------------------------------------------------------------
void Memory_Memset( unsigned char idata *pRam, unsigned char mChar, unsigned char mLen )
{
    while( mLen-- ){
        *pRam++ = mChar;
    }
}
```

H 文件 Memory.h

```
#ifdef _MemoryH
    //用户可以使用的变量

#else
    //-----------------------------------------------------------
    //内存初始化
    //内存初试指针总是指向 RAM 最后
    //-----------------------------------------------------------
    extern void Memory_Init(unsigned char idata *pRamEnd);

    //---------------------------------------------------------------------------
    //析构
    //---------------------------------------------------------------------------
    extern void Memory_Destory(void);

    //---------------------------------------------------------------------------
    //内存分配
    //返回分配后的起始指针
    //---------------------------------------------------------------------------
    extern unsigned char * Memory_Malloc( unsigned char mSize );

    //---------------------------------------------------------------------------
    //用指定数据清除内存
    //---------------------------------------------------------------------------
    extern void Memory_Memset( unsigned char idata *pRam, unsigned char mChar, unsigned char mLen );

#endif
```

程序说明：

一个模块文件，必须包含 x_Init()初始化函数和 x_Destory()析构函数，分别在初始化时候被主程序显式调用、在进入 IDLE、STOP 状态时释放硬件资源用。即使没有相关动作也必须有空函数，保证程序结构的完整性。

这样做的好处：可以避免模块使用到的变量没有清零或赋值，而这是非常容易被遗忘而产生 BUG，而且在进入 IDLE、STOP 状态前调用 x_Destory()释放硬件资源，退出 IDLE、STOP 可以重新调用 x_Init()

初始化变量，保证模块可以重新开始运行。

51 系列内部的堆栈是向 FF 生长的，内部 RAM 内存分配是从 FF 向 00 开始的，可以很好保证 C 编译器的工作区域。

Main.c 为：

```
//-----------------------------------------------------------
//主程序
//-----------------------------------------------------------
#include <reg52.h>

#include "Memory.h"

void main(void)
{
    Memory_Init( (unsigned char idata *)0xff );        //内部 RAM 的最后端

    EA = 1;

    while(1)
    {

    }
}
```

这时可以进行编译和测试，保证 Mempry.c 正确。

Memory_Init_Default()是采用内部 RAM 做内存分配，如果使用外部 RAM 做内存分配，可以单独再建立一个模块。

4）编写通用队列模块并添加到项目中。

队列在 MCU 设计中是会大量被采用的，建立一个通用的队列标准模块是非常重要的。所说的通用，是指非常标准化，标准化的程序达到生成标准库的程度。下面就是通用队列模块的实现：

Queue.h 文件：

```
//-------------------------------------------------------------------------
//通用缓冲队列
//-------------------------------------------------------------------------
struct QueueBuffer{
    unsigned char idata *aBufferStart;        //缓冲区起始点
    unsigned char idata *aBufferEnd;     //缓冲区结束点
    unsigned char idata *pIn;                 //写指针
    unsigned char idata *pOut;                //读指针
    unsigned char mCount;                     //缓冲区数据个数
};

#ifdef _QueueH
    //用户可以使用的变量

#else
    //-----------------------------------------------------------
    //构造
    //-----------------------------------------------------------
    extern void Queue_Init(void);

    //-----------------------------------------------------------
    //析构,释放在 init()中使用到的硬件资源
    //-----------------------------------------------------------
    extern void Queue_Destory(void);

    //-----------------------------------------------------------
    //申请注册缓冲队列
    //返回 struct QueueBuffer 结构指针。
    //-----------------------------------------------------------
    extern struct QueueBuffer idata * Queue_Register( unsigned char mSize );

    //-----------------------------------------------------------
    //将数据压入队列
    //-----------------------------------------------------------
    extern void Queue_Push( struct QueueBuffer idata *pQueueBuffer, unsigned char mData );

    //-----------------------------------------------------------
    //将数据弹出队列
    //-----------------------------------------------------------
    extern unsigned char Queue_Pop( struct QueueBuffer idata *pQueueBuffer );

    //-----------------------------------------------------------
    //读出队列指定序号数据
    //-----------------------------------------------------------
```

```
extern unsigned char Queue_Read( struct QueueBuffer idata *pQueueBuffer, unsigned char mId );

//-------------------------------------------------------------
//返回队列数据个数
//-------------------------------------------------------------
extern unsigned char Queue_Num( struct QueueBuffer idata *pQueueBuffer );

//-------------------------------------------------------------
//队列清空
//-------------------------------------------------------------
extern void Queue_Clear( struct QueueBuffer idata *pQueueBuffer );

#endif
```

Queue.c 文件：

```
//-----------------------------------------------------------------------------
//通用缓冲队列
//-----------------------------------------------------------------------------
#define _QueueH

#include <reg52.h>
#include "Memory.h"
#include "Queue.h"



//-----------------------------------------------------------------------------
//构造
//-----------------------------------------------------------------------------
void Queue_Init(void){ }

//-----------------------------------------------------------------------------
//析构,释放在 init()中使用到的硬件资源
//-----------------------------------------------------------------------------
void Queue_Destory(void){ }

//-------------------------------------------------------------
//申请注册缓冲队列
//返回 struct QueueBuffer 结构指针。
//-------------------------------------------------------------
struct QueueBuffer idata * Queue_Register( unsigned int mSize )
{
    struct QueueBuffer idata *pQueueBuffer;

    pQueueBuffer = (struct QueueBuffer idata * )Memory_Malloc( sizeof(struct QueueBuffer) );
//分配记录队列信息的结构变量
    Memory_Memset( (unsigned char idata *)pQueueBuffer, 0, sizeof(struct QueueBuffer) );
//记录队列信息的结构变量清 0

    pQueueBuffer->aBufferStart = pQueueBuffer->pIn = pQueueBuffer->pOut = Memory_Malloc( mSize );
//分配队列所需内存
```

```
        pQueueBuffer->aBufferEnd = pQueueBuffer->aBufferStart + mSize;
//队列结束指针

        return pQueueBuffer;
    }

    #pragma NOAREGS
    //-------------------------------------------------------------
    //将数据压入队列
    //-------------------------------------------------------------
    void Queue_Push( struct QueueBuffer idata * pQueueBuffer, unsigned char mData )
    {
        unsigned char idata *p;
        p = pQueueBuffer->pIn;

        *p++ = mData;
        if (p == pQueueBuffer->aBufferEnd){
            p = pQueueBuffer->aBufferStart;
        }
        pQueueBuffer->pIn = p;

        pQueueBuffer->mCount ++;
    }

    //-------------------------------------------------------------
    //将数据弹出队列
    //-------------------------------------------------------------
    unsigned char Queue_Pop( struct QueueBuffer idata *pQueueBuffer )
    {
        unsigned char mData;
        unsigned char idata *p;
        p = pQueueBuffer->pOut;

        mData = *p;
        if( ++p == pQueueBuffer->aBufferEnd ){
            p = pQueueBuffer->aBufferStart;
        }
        pQueueBuffer->pOut = p;

        pQueueBuffer->mCount --;

        return mData;
    }

    //-------------------------------------------------------------
    //读出队列指定序号数据
    //-------------------------------------------------------------
    unsigned char Queue_Read( struct QueueBuffer idata *pQueueBuffer, unsigned char mId )
    {
        unsigned char idata *pTemp;
```

```
    pTemp = pQueueBuffer->pOut + mId;
    if( pTemp < pQueueBuffer->aBufferEnd )return(*pTemp);
    else return( *(pTemp - pQueueBuffer->aBufferEnd + pQueueBuffer->aBufferStart) );
}

//-------------------------------------------------------------
//返回队列数据个数
//-------------------------------------------------------------
unsigned char Queue_Num( struct QueueBuffer idata *pQueueBuffer )
{
    return pQueueBuffer->mCount;
}
#pragma AREGS

//-------------------------------------------------------------
//队列清空
//-------------------------------------------------------------
void Queue_Clear( struct QueueBuffer idata *pQueueBuffer )
{
    B = IE;                                          //清零阶段禁止中断，防止被使用
    EA = 0;
    pQueueBuffer->pIn = pQueueBuffer->pOut = pQueueBuffer->aBufferStart;
    pQueueBuffer->mCount = 0;
    IE = B;
}
```

程序说明：

此通用队列用来说明结构，指针都采用 unsigned char idata *而不是通常的 unsigned char *,而且队列的 count 变量采用 char 型，编译结果为：*21.4* 字节和 *562* 字节代码。

struct QueueBuffer * Queue_Register(unsigned int mSize)是很关键的函数，因为它才实现了通用化，以后在标准化的设计中会经常看到此类函数。它的作用是负责外部模块的队列注册，外部模块需记录返回的结构指针，并在以后调用队列函数时将此指针传递进去。该函数内部还负责了注册队列时需要的结构变量内存和队列缓冲区大小的内存，这时，前面写的内存管理函数就开始起作用了。

5）系统时间管理模块

该模块生成 20ms 的系统时间标志和闪烁需要的标志，可以提供所有其它程序模块使用。是构成一个系统所必须的模块。20ms 时间标志的特性将在以后继续展示，这里先表明一点：20MS 在一个周期不是非常准确的，但产生的周期从长远眼光看是准确的，比如一天的误差和一个周期的误差是相等的，只不过在一个周期产生时时间在左右摇摆。

系统需要的时间不要求一定 20ms，只要在 15-20ms 左右都可以，只要提供一个基准的时间就可以，这是系统的特性。如果需要精确的时间，此类任务将归纳为实时性质任务，必须采用中断完成，而不是这里的非实时。

该模块使用了 T0 定时器，采用 16 位定时器的模式 1。

SysTimer.h 文件：

```
//-------------------------------------------------------------------------
//系统时间管理
//-------------------------------------------------------------------------
#define T0PRELOAD_22M  144L
#define T0PRELOAD_11M  (T0PRELOAD_22M/2)

#ifdef _SysTimerH
    //用户可以使用的变量
    bit fSysTimer_Touch;
    bit fSysTimer_FlashTouch;
    bit fSysTimer_FlashState;
#else
    extern bit fSysTimer_Touch;              //系统时间触发
    extern bit fSysTimer_FlashTouch;     //闪烁时间触发
    extern bit fSysTimer_FlashState;     //闪烁状态

    //-----------------------------------------------------------
    //构造
    //入口：产生 20MS 需要的 T0 高位预装值、闪烁亮的时间计数、闪烁灭的时间计数
    //-----------------------------------------------------------
    extern void SysTimer_Init( unsigned char mPreLoad, unsigned char mFlashOnCount, unsigned char mFlashOffCount );

    //-----------------------------------------------------------
    //析构,释放在 init()中使用到的硬件资源
    //-----------------------------------------------------------
    extern void SysTimer_Destory(void);

    //-------------------------------------------------------------------------
    //循环
    //TF0=1 时进入
    //-------------------------------------------------------------------------
    extern void SysTimer_Loop(void);

    //main.c 使用的主循环
    #define SysTimer_MainLoop() do{  fSysTimer_Touch = fSysTimer_FlashTouch = 0; if(TF0)SysTimer_Loop(); }while(0)
```

```
    //22.1184M 系统默认参数配置初始化
    #define SysTimer_Init22M_Default() do{ SysTimer_Init(TOPRELOAD_22M, 500/20, 500/20 );}while(0)
    //11.0592M 系统默认参数配置初始化
    #define SysTimer_Init11M_Default() do{ SysTimer_Init(TOPRELOAD_11M, 500/20, 500/20 );}while(0)

#endif
```

SysTimer.c 文件：

```
//-------------------------------------------------------------
//系统时间管理
//-------------------------------------------------------------
#define _SysTimerH

#include <reg52.h>
#include "Memory.h"
#include "SysTimer.h"

struct InSysTimer{
    unsigned char mT0_PreLoad;          //T0 在模式 0 的 13 位模式下的高位初装值，用来产生系统时间用
    unsigned char mFlashOff_Count;          //闪烁灭的时间计数器
    unsigned char mFlashOn_Count;           //闪烁亮的时间计数器
    unsigned char mFlash_Count;         //闪烁计数
};
struct InSysTimer sInSysTimer;

//-------------------------------------------------------------------------
//系统时间初始化
//T0 工作在 16 位的模式 1 下
//入口：产生 20MS 需要的 T0 高位预装值
//-------------------------------------------------------------------------
void SysTimer_Init( unsigned char mPreLoad, unsigned char mFlashOnCount, unsigned char mFlashOffCount )
{
    //变量清 0
    Memory_Memset( (unsigned char *)&sInSysTimer, 0, sizeof(struct InSysTimer) );

    sInSysTimer.mT0_PreLoad = (~mPreLoad) + 1;          //T0 高位预装值
    sInSysTimer.mFlashOn_Count = mFlashOnCount;
    sInSysTimer.mFlash_Count = sInSysTimer.mFlashOff_Count = mFlashOffCount;

    TMOD |= 0x1;                            //T0.M0 = 1;
    TH0 = sInSysTimer.mT0_PreLoad;
    TR0 = 1;                                //T0 开始运行
}

//-------------------------------------------------------------------------
//析构
//-------------------------------------------------------------------------
void SysTimer_Destory(void)
{
    TR0 = 0;
```

```
    TMOD &= ~0x1;                           //释放硬件资源
}

//-----------------------------------------------------------------------
//循环
//TF0=1 时进入
//-----------------------------------------------------------------------
void SysTimer_Loop(void)
{
    TF0 = 0;
    TH0 = sInSysTimer.mT0_PreLoad;        //重装 T0 高位初始值

    fSysTimer_Touch = 1;                  //系统时间到达

    //闪烁时间控制
    if( fSysTimer_FlashState ){
        //处于闪烁亮
        if( --sInSysTimer.mFlash_Count == 0 ){
            //进入闪烁灭的状态
            sInSysTimer.mFlash_Count = sInSysTimer.mFlashOff_Count;
            fSysTimer_FlashState = 0;
            fSysTimer_FlashTouch = 1;
        }
    }
    else {
        //处于闪烁灭
        if( --sInSysTimer.mFlash_Count == 0 ){
            //进入闪烁亮的状态
            sInSysTimer.mFlash_Count = sInSysTimer.mFlashOn_Count;
            fSysTimer_FlashState = 1;
            fSysTimer_FlashTouch = 1;
        }
    }
}
```

这时的 **main.c** 文件如下，可测试系统时间和闪烁时间。

```
//-------------------------------------------------------------
//主程序
//-------------------------------------------------------------
#include <reg52.h>
#include <intrins.h>

#include "Memory.h"
#include "Queue.h"
#include "SysTimer.h"

void main(void)
{
    Memory_Init( (unsigned char idata *)0xff );        //内部 RAM 的最后端
```

```
    Queue_Init();
    SysTimer_Init22M_Default();

    EA = 1;                                                    //开中断，开始运行系统

    while(1)
    {
        SysTimer_MainLoop();

        //测试系统时间和闪烁时间
        if( fSysTimer_Touch ){
            _nop_();
        }

        if( fSysTimer_FlashTouch ){
            if( fSysTimer_FlashState ){
                _nop_();
            }
            else {
                _nop_();
            }
        }
    }
}
```

在三个_nop_();处设置断点，可观察到运行的时间值。第一个每 20ms 停顿，第二、第三每 500ms 停顿。

6） 消息管理模块

消息管理在模块标准化的设计中占有举足轻重的地位，它是所有模块级别之间信息交流的桥梁，有了消息管理模块，极其多的程序或模块可以做成一个通用的标准模块，然后你可以将它们做成一个标准库文件，极大地减少了重复性开发的工作。有了消息管理，可以保证模块的完全独立性，这种完全独立性的实现才可以称之为**模块化**！

模块化的概念：

平时常说的“模块化”是一个非常空洞的词语，不是简单的把功能放在一个 C 文件就可以的，而是把所有函数、变量全部封装起来，和外部其它模块脱离联系，这种程度的模块才称为模块化。做到这种程度，甚至还可以在程序运行中将模块卸载掉而只影响本身功能，不影响整个系统和其它功能。模块化后还有一个非常大的好处：具备了真正意义上的测试控制。可以通过消息的交流、拦截、转义对模块进行综合性质的测试，而且在模块功能完全不改变的情况下实现。

在前面几个通用系统函数建立后，消息管理模块的基础就已经具备了。

消息的结构：

消息管理实际上是对一个消息队列的管理，只不过增多了一些特征：

1） 定义了消息的协议规范。（消息是可以带参数的。）
2） 中断中只能使用消息模块中的压入（push）函数，不能使用队列首部插入(Insert)函数。
3） 中断中也不能弹出消息。消息处理属于非实时概念和领域。
4） 非中断下可以队列首部插入(Insert)函数，提前处理，保证要处理的信息可以连贯操作。
5） 对无主的消息进行了自我处理。
6） 每个消息是一个枚举量，即一个独立的数字，它对应着一个要被处理的函数指针。

消息的协议规范：

消息字 + 参数字节长度 + 参数段。

比如：

要压入一个不带参数的消息，依次会压入 eMsg_xx, 0

要压入一个带一个字节 0xf 的消息，依次会压入 eMsg_xx, 1,0xf

Message.h 文件：

```
//-----------------------------------------------------------------------
//消息管理
//-----------------------------------------------------------------------

//消息字定义，以 eMSG_End 结尾
enum MSG_HEAD{
    eMsg_PreProcess,                    //这是消息中心的消息，通常为 MenuNewsProcess，被第一个处理作为消息转换
    eMsg_LagProcess,                    //消息滞后处理，一般菜单调用，作为消息未接受时处理。

    //用户自定义消息字.
    #include "User_Message.txt"         //这是用户定义要使用的消息字枚举文件

    eMSG_End                            //消息结尾，系统处理
};

//-----------------------------------------------------------------------
#ifdef _MessageH
```

```
#else
    //初始化。入口：消息队列的最大数
    extern void Msg_Init( unsigned int mMsgBufferSize, unsigned char mMsgMax );

    //析构,释放在 init()中使用到的硬件资源
    extern void Msg_Destory(void);

    //消息处理程序的登记。入口：消息字,函数指针
    extern void Msg_Register(void ( *Function)(), unsigned char mEnumMin, unsigned char mEnumMax );

    //=============================================================================
    //弹出数据.不能在中断中使用
    extern unsigned char Msg_PopMsg(void);
    extern unsigned char Msg_PopByte(void);
    extern unsigned int Msg_PopInt(void);

    //压入消息
    extern void Msg_Push( unsigned char mMsg );
    extern void Msg_PushByte( unsigned char mMsg, unsigned char mData );
    extern void Msg_PushInt( unsigned char mMsg, unsigned int mData );

    //在队列头部插入消息
    extern void Msg_Insert( unsigned char mMsg );
    extern void Msg_InsertByte(unsigned char mMsg,unsigned char mData);
    extern void Msg_InsertInt(unsigned char mMsg,unsigned int mData);

    //消息循环，负责发送消息和无任何程序接收消息时销毁消息
    extern void Msg_Loop(void);

    //主循环
    #define Msg_MainLoop()  do{ Msg_Loop(); }while(0)
#endif
```

Message.c 文件：

```
//-----------------------------------------------------------------------
//消息管理
//消息结构：枚举型的消息字 + 参数个数 + 参数段
//-----------------------------------------------------------------------
#define _MessageH

#include <reg52.h>
#include "Memory.h"
#include "Message.h"

struct InMsg{
    void (**aMessage_Function)();                   //对应消息字的函数指针数组开始指针

    unsigned char idata *aMsgBufStart;              //消息缓冲区开始
    unsigned char idata *aMsgBufEnd;           //消息缓冲区结束位置
```

```
    unsigned char idata *pMsgIn;
    unsigned char idata *pMsgOut;

    unsigned char aReadMsgBuffer[5];          //用户使用的消息信息缓冲区，只读.[0]消息字，[1]-[4]为
参数?
};
data struct InMsg sInMsg;

#pragma NOAREGS
//---------------------------------------------------------------------------
//弹出 char 数据.
//---------------------------------------------------------------------------
unsigned char InMsg_Pop(void)
{
    unsigned char mData;

    mData = *sInMsg.pMsgOut;
    if( sInMsg.pMsgOut == sInMsg.aMsgBufEnd )
    {
        sInMsg.pMsgOut = sInMsg.aMsgBufStart;
    }
    else sInMsg.pMsgOut++;

    return mData;
}

//---------------------------------------------------------------------------
//内部函数：拷贝消息到消息缓存
//---------------------------------------------------------------------------
void InMsg_CopyToBuffer(void)
{
    unsigned char i;
    unsigned char idata *p = (unsigned char idata *)&sInMsg.aReadMsgBuffer[1];

    sInMsg.aReadMsgBuffer[0] = InMsg_Pop();
    i = InMsg_Pop();                              //消息长度

    while( i-- ){
        *p++ = InMsg_Pop();
    }

}

//---------------------------------------------------------------------------
//内部压入数据到队列尾
//---------------------------------------------------------------------------
void InMsg_Push_Tail( unsigned char mData )
{
    *sInMsg.pMsgIn = mData;
    if( sInMsg.pMsgIn == sInMsg.aMsgBufEnd )
```

```
        {
            sInMsg.pMsgIn = sInMsg.aMsgBufStart;
        }
        else sInMsg.pMsgIn++;
    }

    //-------------------------------------------------------------------------
    //内部压入数据到队列头
    //-------------------------------------------------------------------------
    void InMsg_Insert_Head( unsigned char mData )
    {
        if( sInMsg.pMsgOut == sInMsg.aMsgBufStart )
        {
            sInMsg.pMsgOut = sInMsg.aMsgBufEnd;
        }
        else sInMsg.pMsgOut --;
        *sInMsg.pMsgOut = mData;
    }
    #pragma AREGS
    //=========================================================================================================
=

    //-------------------------------------------------------------------------
    //构造函数
    //-------------------------------------------------------------------------
    void Msg_Init( unsigned int mMsgBufferSize, unsigned char mMsgMax )
    {
        Memory_Memset( (unsigned char idata *)&sInMsg, 0, sizeof(struct InMsg) );

        //申请消息缓冲区
        sInMsg.aMsgBufStart = sInMsg.pMsgIn = sInMsg.pMsgOut = (unsigned char idata
*)Memory_Malloc( mMsgBufferSize );
        sInMsg.aMsgBufEnd = sInMsg.aMsgBufStart + mMsgBufferSize - 1;

        //消息处理函数数组分配
        sInMsg.aMessage_Function = (void *)Memory_Malloc( sizeof(void *) * mMsgMax );
    }

    //-------------------------------------------------------------------------
    //析构,释放在 init()中使用到的硬件资源
    //-------------------------------------------------------------------------
    void Msg_Destory(void){ }

    //-------------------------------------------------------------------------
    //消息处理程序的登记
    //入口：消息字,函数指针
    //开始消息字和结束消息字之间的处理函数指针设置
    //-------------------------------------------------------------------------
    void Msg_Register(void ( *Function)(), unsigned char mEnumMin, unsigned char mEnumMax )
    {
```

```
        while( mEnumMin <= mEnumMax )
        {
            sInMsg.aMessage_Function[ mEnumMin++ ] = Function;
        }
    }

    //=============================================================================================
=
    //-----------------------------------------------------------------------
    //弹出消息.
    //不能在中断中使用
    //-----------------------------------------------------------------------
    unsigned char Msg_PopMsg(void)
    {
        register unsigned char i;

        i = sInMsg.aReadMsgBuffer[0] ;
        sInMsg.aReadMsgBuffer[0] = 0;                   //已经处理
        return i;
    }

    //-----------------------------------------------------------------------
    //弹出 Byte 数据.
    //不能在中断中使用
    //-----------------------------------------------------------------------
    unsigned char Msg_PopByte(void)
    {
        return sInMsg.aReadMsgBuffer[1] ;
    }

    //-----------------------------------------------------------------------
    //弹出 Int 数据.
    //不能在中断中使用
    //-----------------------------------------------------------------------
    unsigned int Msg_PopInt(void)
    {
        return *( (unsigned int *)&sInMsg.aReadMsgBuffer[1] );
    }



    //=============================================================================================
=
    #pragma NOAREGS
    //-----------------------------------------------------------------------
    //压入消息,不带参数
    //-----------------------------------------------------------------------
    void Msg_Push( unsigned char mMsg )
    {
        B = IE;
        EA = 0;
```

```
    InMsg_Push_Tail( mMsg );                    //消息字
    InMsg_Push_Tail( 0 );                       //消息长度
    IE = B;
}

//-------------------------------------------------------------------------
//压入消息,带一个字节参数
//-------------------------------------------------------------------------
void Msg_PushByte( unsigned char mMsg, unsigned char mData )
{
    B = IE;
    EA = 0;
    InMsg_Push_Tail( mMsg );                    //消息字
    InMsg_Push_Tail( 1 );                       //消息长度
    InMsg_Push_Tail( mData );                   //参数
    IE = B;
}

//-------------------------------------------------------------------------
//压入消息,带一个 int 参数
//-------------------------------------------------------------------------
void Msg_PushInt( unsigned char mMsg, unsigned int mData )
{
    B = IE;
    EA = 0;
    InMsg_Push_Tail( mMsg );                    //消息字
    InMsg_Push_Tail( 2 );                       //消息长度
    InMsg_Push_Tail( mData/256 );               //参数高位。*C51 是高位在前，和其它 C 是反的!
    InMsg_Push_Tail( mData );                   //参数
    IE = B;
}

//=========================================================================================
=

//-------------------------------------------------------------------------
//在队列头部插入消息,不带参数
//-------------------------------------------------------------------------
void Msg_Insert( unsigned char mMsg )
{
    B = IE;
    EA = 0;
    InMsg_Insert_Head( 0 );                     //消息长度
    InMsg_Insert_Head( mMsg );                  //消息字
    IE = B;
}


//-------------------------------------------------------------------------
//向消息队列头插入一条带一个参数的消息
```

```
//入口：消息字，参数
//---------------------------------------------------------------------------
void Msg_InsertByte(unsigned char mMsg,unsigned char mData)
{
    B = IE;
    EA = 0;
    InMsg_Insert_Head(mData);
    InMsg_Insert_Head(1);                                   //压入参数长度
    InMsg_Insert_Head(mMsg);                                //压入消息字
    IE = B;
}

//---------------------------------------------------------------------------
//向消息队列头插入一条带一个 int 的消息
//入口：消息字，int
//---------------------------------------------------------------------------
void Msg_InsertInt(unsigned char mMsg,unsigned int mData)
{
    B = IE;
    EA = 0;
    InMsg_Insert_Head(mData/256);
    InMsg_Insert_Head(mData);
    InMsg_Insert_Head(2);                                   //压入参数长度
    InMsg_Insert_Head(mMsg);                                //压入消息字
    IE = B;
}
#pragma AREGS
//=============================================================================================================
=

//---------------------------------------------------------------------------
//消息循环，负责发送消息和无任何程序接收消息时销毁消息
//---------------------------------------------------------------------------
void Msg_Loop(void)
{
    if( sInMsg.aReadMsgBuffer[0] )
    {
        //消息没有被任何一个程序接受，取消
        if( sInMsg.aMessage_Function[ eMsg_LagProcess ] )
        {
            //存在外部消息滞后处理函数
            ( *sInMsg.aMessage_Function[ eMsg_LagProcess ] )();
        }
        else
        {
            //取消,无效消息
            sInMsg.aReadMsgBuffer[0] = 0;
        }
    }
```

```
        //消息发放
        if( sInMsg.pMsgIn != sInMsg.pMsgOut )
        {
            //拷贝到消息缓存
            InMsg_CopyToBuffer();

            //检查是否需要消息预处理（菜单调用）
            if( sInMsg.aMessage_Function[ eMsg_PreProcess ] )
            {
                ( *sInMsg.aMessage_Function[ eMsg_PreProcess ] )();
            }

            //调用消息处理对象函数
            if( (sInMsg.aReadMsgBuffer[0]) && ( sInMsg.aMessage_Function[ sInMsg.aReadMsgBuffer[0] ] ) )
            {
                ( *sInMsg.aMessage_Function[ sInMsg.aReadMsgBuffer[0] ] )();
            }
        }
    }
```

程序说明：

消息枚举中包含两个枚举：eMsg_PreProcess、eMsg_LagProcess，分别是消息在系统处理前调用的外部处理函数和消息在没有被任何模块处理掉的外部处理函数。有这两个函数的支持，就可以建立基于消息管理程序的菜单模块，也可以用来做测试时的消息拦截和转换。要使用它们，可以按平时的消息注册一样，把它们对应的枚举和函数指针做变量传递进去就可以了。比如：

Msg_Register（ My_PreProcessMsg, eMsg_PreProcess, eMsg_PreProcess）;

Msg_Register（ My_LagProcessMsg, eMsg_LagProcess, eMsg_LagProcess）;

或者使用同一函数处理这两个消息：

Msg_Register（ My_ProcessMsg, **eMsg_PreProcess**, **eMsg_LagProcess**）;

关于这个部分，将在新的关于图形系统设计的培训中讲到，这里不多描述。

在中断中，只能使用 Msg_Pushxxx()之类的向消息队列尾压入的函数，这样可以将实时产生的消息转为非实时的消息处理。**消息存在消息队列中被依次顺序处理。**在消息字需要转换或演变时，必须采用 Msg_Insertxxx()之类函数向消息队列的首部插入消息，让消息系统在主循环中能立刻发送，这样可以保证处理逻辑的正确性。比如：刷新屏幕时，需要通知相关模块刷新相关区域，这时立刻向自己和相关模块插入刷新的消息，不能向消息尾部压入，这样在状态改变前能全部刷新完要刷新的，否则消息压入尾部被运行到时，屏幕状态逻辑可能发生变化，这样通常会产生 BUG。插入消息，可以实现消息的再次细化或转义。

用户使用的消息字定义，放在 User_Message.txt 文件中，是消息字枚举的一个部分。

这时候的 main.c 文件：

```
//-------------------------------------------------------------
//主程序
//-------------------------------------------------------------
#include <reg52.h>

#include "Memory.h"
#include "Queue.h"
#include "SysTimer.h"
```

```
#include "Message.h"

void main(void)
{
    Memory_Init( (unsigned char idata *)0xff );         //内部 RAM 的最后端
    Queue_Init();
    Msg_Init(20);                                       //使用 20 字节的消息缓冲区

    SysTimer_Init22M_Default();

    EA = 1;                                             //开中断，开始运行系统

    while(1)
    {
        SysTimer_MainLoop();
        Msg_MainLoop();
    }
}
```

这时观察 main.c 可以发现：

只要写完一个模块，我们只要在 main.c 中增加一个初始化函数和主循环函数就可以开始模块的工作，这是非常重要的**模块化特征**。不管系统有多么的庞大，多么复杂的模块功能，它们的对外接口都是非常简单的，加入到 main.c 中开始工作也只要这两个函数就可以！

到此，需要构成一个通用系统的基本要求框架已经建立起来了，我们可以将上面的这些模块全部生成一个标准库，这样我们在使用 51 时就能重复使用这些资源，而且还能建立规范标准的程序，能大量减少重复性开发的工作。

下面，开始新的一章：通用函数的建立。包括：按键处理通用模块、通用数码显示模块。讲述了建立通用函数的规范和思想。*通用函数是指能生成标准库的函数。在后续培训中陆续增加其它通用函数。

7） 设计通用按键扫描模块

这个模块在基于前面提到的消息管理模块的基础上写的，封装了对按键扫描过程的处理和按键消息的转义（扩展快速按键、长按键等消息），它采用回调函数的方式调用了外部和硬件有关的按键扫描码函数。它可以被生成为一个标准库。

按键可以重复发生，保持一个按键按下，其它按键发生按下同样还可以再发出单独的对应按键消息，这是一个重大的突破。

首先必须在 User_Message.txt 中添加和按键有关的消息字，这里有 8 个按键，这时 User_Message.txt 中内容如下：

```
//----------------按键消息字------------------
Msg_Key0,
Msg_Key1,
Msg_Key2,
Msg_Key3,
Msg_Key4,
Msg_Key5,
Msg_Key6,
Msg_Key7,
//---------------按键消息字结束----------------
```

KeySwap.h 文件：

```
//-------------------------------------------------------------
//系统键盘扫描控制部分
//
//-------------------------------------------------------------

//-------------------------------------------------------------
//按键消息包含的信息
enum KeyMsgUnit{
    eKeyMsg_Down,                                   //按键按下,基本信息
    eKeyMsg_Up,                                     //按键弹起，基本信息
    eKeyMsg_LongPush,                               //长按，在外部程序处理
    eKeyMsg_SpeedTwice,                             //快速两次按键，在外部程序处理
    eKeyMsg_NormalPush                              //正常按键
};



#define KEYTWICEDELAY       20                      //0.4s,两次快速间隔按键作为键虚拟，间隔时间<0.4s
#define KEYLONGDELAYMAX     80                      //1.6s/20ms = 84，>1.6S 做长按键处理

//-------------------------------------------------------------
//转换按键消息，处理长按键、正常按键、快速两次按键用的结构
struct Key_Translate
{
    unsigned char mDelay;                           //按键保持延时计数
    unsigned char mKeyState;                        //按键状态
};

struct KeySwap{
```

```
        unsigned char idata *pKeyState;                          //根据按键数目在初始化动态分配的 KEY 状态位数组开始指针
    };

    //-----------------------------------------------------------
    #ifdef _KeySwapH
        //定义用户使用的变量
        struct KeySwap sKeySwap;

    #else
        extern struct KeySwap sKeySwap;

        //-----------------------------------------------------------
        //初始化
        //-----------------------------------------------------------
        extern void KeySwap_Init(void);

        //-----------------------------------------------------------
        //析构
        //-----------------------------------------------------------
        extern  void KeySwap_Destory(void);

        //-----------------------------------------------------------
        //使用注册
        //-----------------------------------------------------------
        extern void KeySwap_Register( void ( *Function)(), unsigned char mKeyMsgStart, unsigned char mKeyNum );

        //-----------------------------------------------------------
        //循环
        //系统时间进入一次
        //-----------------------------------------------------------
        extern  void KeySwap_Loop(void);

        //-----------------------------------------------------------
        //转换按键消息，处理长按键、正常按键、快速两次按键
        //入口：要处理的按键处理结构，消息带的按键状态的参数
        //被外部模块调用
        //-----------------------------------------------------------
        extern  void KeySwap_Translate( struct Key_Translate *pBase_Key_Translate, unsigned char mKeyMsg, unsigned char mParameter);

        //-----------------------------------------------------------
        //转换按键消息外部调用循环
        //入口：要处理的按键处理结构，对应的按键消息
        //被外部模块调用
        //-----------------------------------------------------------
        extern void KeySwap_TranslateLoop( struct Key_Translate *Key_Translate, unsigned char mKeyMsg);

    #endif
```

KeySwap.c 文件：

```
//-------------------------------------------------------------
//系统键盘扫描控制部分
//-------------------------------------------------------------
#define _KeySwapH

#include <reg52.h>
#include "Memory.h"
#include "Message.h"

#include "KeySwap.h"

//-------------------------------------------------------------
enum eBase_KeyTask{
    eBase_Idle,                                     //空闲任务
    eBase_PushDelay,                                //按键按下判断延时
    eBase_PushCheck,                                //按键按下判断
    eBase_PopDelay,                                 //按键松开判断延时
    eBase_PopCheck                                  //按键松开判断
};

enum eBase_KeyState{
    eBase_KeyNoMove,                                //按键未动作
    eBase_KeyPushMove,                              //按下动作
    eBase_KeyPopMove                                //松开动作
};

//-------------------------------------------------------------
//按键消息转换使用的状态
enum InKeyTranslate{
    eKeyTran_Idle,                                  //初始状态
    eKeyTran_Time,                                  //按键计时
    eKeyTran_SpeedTwiceWait,
    eKeyTran_SpeedTwicePush,                        //快速两次按键，等待按键释放
    eKeyTran_LongPush                               //长按键状态，等待按键释放
};

//-------------------------------------------------------------
struct InKeySwap{
    unsigned char idata *pKeyState_Stick;           //保留按键状态，做判别用。和 pKeyState 对应,
暂存器。
    unsigned char mKeyByteNum;                      // 按 键 个 数 -> 按 键 字 节 数 ,pKeyState 、
pKeyState_Stick 字节大小。
    unsigned char mKeyProcByte;                     //正在处理的 KEY 对应的字节
    unsigned char mKeyProcBit;                      //正在处理的 KEY 对应的位

    unsigned char mKeyMsgStart;                     //按键消息开始
    void (*CallBack_KeySwap)();                     //扫描键值回调函数指针

    unsigned char mTask;                            //扫描的状态变化任务号
```

```
};
struct InKeySwap sInKeySwap;                                    //内部使用变量

//-------------------------------------------------------------
//初始化
//-------------------------------------------------------------
void KeySwap_Init(void)
{
     Memory_Memset( (unsigned char *)&sInKeySwap, 0, sizeof(struct InKeySwap) );
     Memory_Memset( (unsigned char *)&sKeySwap, 0, sizeof(struct KeySwap) );
}

//-------------------------------------------------------------
//使用注册。根据按键数分配要使用到的 RAM。
//入口：扫描码回调函数指针，按键消息字的开始，按键数目
//-------------------------------------------------------------
void KeySwap_Register( void ( *Function)(), unsigned char mKeyMsgStart, unsigned char mKeyNum )
{
     sInKeySwap.mKeyMsgStart = mKeyMsgStart;                        //按键消息的起始
     sInKeySwap.CallBack_KeySwap = Function;

     sInKeySwap.mKeyByteNum = ((mKeyNum-1)>>3) + 1;          //按键个数->按键字节数

     //分配内存
     sKeySwap.pKeyState = Memory_Malloc( sInKeySwap.mKeyByteNum );
     Memory_Memset( sKeySwap.pKeyState, 0, sInKeySwap.mKeyByteNum );

     sInKeySwap.pKeyState_Stick = Memory_Malloc( sInKeySwap.mKeyByteNum );
     Memory_Memset( sInKeySwap.pKeyState_Stick, 0, sInKeySwap.mKeyByteNum );
}

//-------------------------------------------------------------
//析构,释放在 init()中使用到的硬件资源
//-------------------------------------------------------------
void KeySwap_Destory(void){ }

#pragma NOAREGS
//-------------------------------------------------------------
//判断改变的字节中最先遇到的改变位
//-------------------------------------------------------------
unsigned char InKeySwap_StateBit(unsigned char mData0, unsigned char mData1)
{
     unsigned char i;
     unsigned char mLogic = 0x1;

     for( i=0; i<7; i++){
          if( (mData0&mLogic) != (mData1&mLogic) )return mLogic;
          mLogic <<= 1;
     }
     return 0x80;
```

```
}

//-------------------------------------------------------------
//判断按键状态
//返回：
//   eBase_KeyNoMove
//   eBase_KeyPushMove
//   eBase_KeyPopMove
//-------------------------------------------------------------
unsigned char InKeySwap_State(void)
{
    unsigned char i;

    for( i=0; i<sInKeySwap.mKeyByteNum; i++){
        if( sKeySwap.pKeyState[i] != sInKeySwap.pKeyState_Stick[i]){
            sInKeySwap.mKeyProcByte = i;                    //记录字节位置
            sInKeySwap.mKeyProcBit       =       InKeySwap_StateBit(       sKeySwap.pKeyState[i],
sInKeySwap.pKeyState_Stick[i] );
            if( sInKeySwap.pKeyState_Stick[i] & sInKeySwap.mKeyProcBit )
            {
                return eBase_KeyPopMove;                    //原来在按下状态，发生按键松开改变
            }
            else {
                return eBase_KeyPushMove;                   //原来在松开状态，发生按键按下改变
            }
        }
    }
    return eBase_KeyNoMove;                                 //无变化
}

//-------------------------------------------------------------
//取得当前的按键对应的消息字
//-------------------------------------------------------------
unsigned char InKeySwap_Msg(void)
{
    unsigned char i;
    unsigned char mLogic;

    //位转换为个数
    mLogic = sInKeySwap.mKeyProcBit;
    i = 0xff;

    do{
        mLogic >>= 1;
        i++;
    }while( mLogic );

    return sInKeySwap.mKeyMsgStart + (sInKeySwap.mKeyProcByte<<3) + i;
}
```

```
//-------------------------------------------------------------
//循环
//系统时间进入一次，判断扫描码有无改变并做相应处理
//-------------------------------------------------------------
void KeySwap_Loop(void)
{
    if( sInKeySwap.CallBack_KeySwap == 0)return;            //无外部扫描码函数，退出
    (*sInKeySwap.CallBack_KeySwap)();                       //取得扫描码

    switch( sInKeySwap.mTask )
    {
        case eBase_Idle:                                    //空闲任务，开始扫描按键
            switch( InKeySwap_State() ){
                case eBase_KeyNoMove:                       //按键未动作
                    break;

                case eBase_KeyPushMove:                     //按下动作
                    sInKeySwap.mTask = eBase_PushDelay;
                    break;

                case eBase_KeyPopMove:                      //松开动作
                    sInKeySwap.mTask = eBase_PopDelay;
                    break;
            }
            break;

        case eBase_PushDelay:                               //按键按下判断延时
            if(   (sKeySwap.pKeyState[sInKeySwap.mKeyProcByte]   &   sInKeySwap.mKeyProcBit)   !=
(sInKeySwap.pKeyState_Stick[sInKeySwap.mKeyProcByte] & sInKeySwap.mKeyProcBit) )
            {
                sInKeySwap.mTask = eBase_PushCheck;         //继续不等，按键有效
            }
            else{
                sInKeySwap.mTask = eBase_Idle;              //等，为抖动。
            }
            break;

        case eBase_PushCheck:                               //按键按下判断
            if(   (sKeySwap.pKeyState[sInKeySwap.mKeyProcByte]   &   sInKeySwap.mKeyProcBit)   !=
(sInKeySwap.pKeyState_Stick[sInKeySwap.mKeyProcByte] & sInKeySwap.mKeyProcBit) )
            {
                sInKeySwap.pKeyState_Stick[sInKeySwap.mKeyProcByte] |= sInKeySwap.mKeyProcBit;
//保存缓冲寄存器
                Msg_PushByte( InKeySwap_Msg(), eKeyMsg_Down );//按键对应的消息字+Down
            }
            sInKeySwap.mTask = eBase_Idle;                  //一个完整处理过程完毕
            break;

        case eBase_PopDelay:                                //按键松开判断延时
            if(   (sKeySwap.pKeyState[sInKeySwap.mKeyProcByte]   &   sInKeySwap.mKeyProcBit)   !=
```

```
(sInKeySwap.pKeyState_Stick[sInKeySwap.mKeyProcByte] & sInKeySwap.mKeyProcBit) )
                    {
                        sInKeySwap.mTask = eBase_PopCheck;              //继续不等，按键有效
                    }
                    else {
                        sInKeySwap.mTask = eBase_Idle;                  //等，为抖动。
                    }
                    break;

                case eBase_PopCheck:                                    //按键松开判断
                    if(   (sKeySwap.pKeyState[sInKeySwap.mKeyProcByte]   &   sInKeySwap.mKeyProcBit)   !=
(sInKeySwap.pKeyState_Stick[sInKeySwap.mKeyProcByte] & sInKeySwap.mKeyProcBit) )
                    {
                        sInKeySwap.pKeyState_Stick[sInKeySwap.mKeyProcByte]  &=  ~sInKeySwap.mKeyProcBit;
//保存缓冲寄存器
                        Msg_PushByte( InKeySwap_Msg(), eKeyMsg_Up );//按键对应的消息字+Up
                    }
                    sInKeySwap.mTask = eBase_Idle;                      //一个完整处理过程完毕
                    break;
            }
        }
        #pragma AREGS

        //++++++++++++++++++++++++++++++++++++     按     键     功     能     扩     展     部     分
++++++++++++++++++++++++++++++++++++++++++++++++++

        //------------------------------------------------------------
        //转换按键消息，自动生成长按键、正常按键、快速两次按键中的一个
        //入口：要处理的按键处理结构，消息带的按键状态的参数
        //该函数被调用后，KeySwap_TranslateLoop 循环中的时间>0,功能被启动。
        //被外部调用
        //------------------------------------------------------------
        void KeySwap_Translate( struct Key_Translate *pKey_Translate, unsigned char mKeyMsg, unsigned char
mParameter)
        {
            switch( pKey_Translate->mKeyState )
            {
                case eKeyTran_Idle:
                    if( mParameter == eKeyMsg_Down ){
                        //开始判断按下时间长度
                        pKey_Translate->mDelay = KEYLONGDELAYMAX;
                        pKey_Translate->mKeyState = eKeyTran_Time;
                    }
                    break;

                case eKeyTran_Time:                                     //      一      定      为
eKeyMsg_Up,pBase_Key_Translate->mDelay 一定>0,所以一定只能快速两次按键或正常按键
                    //判断是否快速两次按键
                    pKey_Translate->mDelay = KEYTWICEDELAY;
                    pKey_Translate->mKeyState = eKeyTran_SpeedTwiceWait;
```

```
                    break;

                case eKeyTran_SpeedTwiceWait:                    //          一          定          为
eKeyMsg_Down,pBase_Key_Translate->mDelay 一定>0,快速两次按键
                    pKey_Translate->mDelay = 0;
                    Msg_InsertByte( mKeyMsg, eKeyMsg_SpeedTwice );
                    pKey_Translate->mKeyState = eKeyTran_SpeedTwicePush;
                    break;

                case eKeyTran_SpeedTwicePush:
                case eKeyTran_LongPush:
                    if( mParameter == eKeyMsg_Up ){
                        pKey_Translate->mKeyState = eKeyTran_Idle;
                    }
                    break;

            }
        }

        //-------------------------------------------------------------
        //转换按键消息外部调用循环
        //入口：要处理的按键处理结构，对应的按键消息
        //被外部调用
        //-------------------------------------------------------------
        void KeySwap_TranslateLoop( struct Key_Translate *pKey_Translate, unsigned char mKeyMsg)
        {
            if( pKey_Translate->mDelay ){
                if( --pKey_Translate->mDelay == 0){
                    //时间到
                    switch( pKey_Translate->mKeyState )
                    {
                        case eKeyTran_Time:                    //判断长按键
                            Msg_InsertByte( mKeyMsg, eKeyMsg_LongPush );
                            pKey_Translate->mKeyState = eKeyTran_LongPush;
                            break;

                        case eKeyTran_SpeedTwiceWait:    //判断快速两次按键还是正常按键。时间到则为正常按
键
                            Msg_InsertByte( mKeyMsg, eKeyMsg_NormalPush );
                            pKey_Translate->mKeyState = eKeyTran_Idle;
                            break;

                        default:
                            break;
                    }

                }
            }
        }
```

这时的 main.c:

```
//-----------------------------------------------------------
//主程序
//-----------------------------------------------------------
#include <reg52.h>

#include "Memory.h"
#include "Queue.h"
#include "SysTimer.h"
#include "Message.h"

#include "KeySwap.h"


void main(void)
{
    //系统模块初始化
    Memory_Init( (unsigned char idata *)0xff );        //内部 RAM 的最后端
    Queue_Init();
    Msg_Init(20);                                      //使用 20 字节的消息缓冲区
    SysTimer_Init22M_Default();

    //通用函数初始化
    KeySwap_Init();

    EA = 1;                                            //开中断，开始运行系统

    while(1)
    {
        SysTimer_MainLoop();
        Msg_MainLoop();
        KeySwap_MainLoop();
    }
}
```

加入按键测试功能：KeyTest

KeyTest.h：

```
//-----------------------------------------------------
//按键功能测试
//假设按键接在 P3.0 和 P3.1 上。对地。
//User_Message.txt 已经加入 Msg_Key0, Msg_Key1,
//第一个 KEY 按下 P3.2=0
//第一个 KEY 长按下 P3.3=0
//-----------------------------------------------------



//-------------------------------------------------
//初始化
//-------------------------------------------------
extern void KeyTest_Init();
```

```
//-------------------------------------------------
//析构
//-------------------------------------------------
extern void KeyTest_Destory();

//-------------------------------------------------
//循环
//20ms 执行一次
//-------------------------------------------------
extern void KeyTest_Loop();

#define KeyTest_MainLoop() do{ if(fSysTimer_Touch)KeyTest_Loop(); }while(0)
```

KeyTest.c：

```
//-------------------------------------------------------
//按键功能测试
//假设按键接在 P3.0 和 P3.1 上。对地。
//User_Message.txt 已经加入 Msg_Key0, Msg_Key1,
//第一个 KEY 按下 P3.2=0
//第二个 KEY 长按下 P3.3=0
//第二个 KEY 快速两次按下 P3.3=1
//-------------------------------------------------------
#include <reg52.h>

#include "Memory.h"
#include "Queue.h"
#include "SysTimer.h"
#include "Message.h"

#include "KeySwap.h"

sbit P3_0 = P3^0;
sbit P3_1 = P3^1;
sbit P3_2 = P3^2;
sbit P3_3 = P3^3;

struct Key_Translate sKeyTest_Translate[1];             //只一个按键需要按键转义

//-------------------------------------------------
//获得扫描码
//*pKeyState 中 0 表示按键松开
//-------------------------------------------------
void InKeyTest_SwapCallBack()
{
    sKeySwap.pKeyState[0] = 0;

    if( !P3_0 )sKeySwap.pKeyState[0] |= 0x1;
    if( !P3_1 )sKeySwap.pKeyState[0] |= 0x2;
```

```
}

//-------------------------------------------------
//按键消息处理函数
//-------------------------------------------------
void InKeyTest_MsgProc()
{
    switch( Msg_PopMsg() )
    {
        case Msg_Key0:    //第一个按键
            switch( Msg_PopByte() )//按键动作
            {
                case eKeyMsg_Down:     //按键按下,基本信息
                    P3_2 = 0;
                    break;

                case eKeyMsg_Up:       //按键弹起，基本信息
                case eKeyMsg_LongPush: //长按，在外部程序处理
                case eKeyMsg_SpeedTwice://快速两次按键，在外部程序处理
                case eKeyMsg_NormalPush://正常按键
                    break;
            }
            break;

        case Msg_Key1:    //第二个按键
            switch( Msg_PopByte() )//按键动作
            {
                case eKeyMsg_Down:     //按键按下,基本信息
                case eKeyMsg_Up:       //按键弹起，基本信息
                    //开始按键消息转义,只对 eKeyMsg_Down、eKeyMsg_Up 这两个基本按键消息起作用
                    KeySwap_Translate( &sKeyTest_Translate[0], Msg_Key1, Msg_PopByte() );
                    break;

                case eKeyMsg_LongPush: //长按，在外部程序处理
                    P3_3 = 0;
                    break;

                case eKeyMsg_SpeedTwice://快速两次按键，在外部程序处理
                    P3_3 = 1;
                    break;

                case eKeyMsg_NormalPush://正常按键
                    break;
             }
             break;
    }
}

//-------------------------------------------------
//初始化
```

```
//-------------------------------------------------
void KeyTest_Init()
{
    Memory_Memset( (unsigned char *)&sKeyTest_Translate[0], 0, sizeof(struct Key_Translate) );
    //注册按键扫描管理
    KeySwap_Register( InKeyTest_SwapCallBack, Msg_Key0, 2 );

    //注册按键消息
    Msg_Register( InKeyTest_MsgProc, Msg_Key0, Msg_Key1 );  //KeyTest_MsgProc() 处 理  Msg_Key0  到 Msg_Key1 之间消息

}

//-------------------------------------------------
//析构
//-------------------------------------------------
void KeyTest_Destory(){ }

//-------------------------------------------------
//循环
//20ms 执行一次
//-------------------------------------------------
void KeyTest_Loop()
{
    KeySwap_TranslateLoop( &sKeyTest_Translate[0], Msg_Key1 );
}
```

测试按键时候的 main.c:

```
//-----------------------------------------------------------
//主程序
//-----------------------------------------------------------
#include <reg52.h>

#include "Memory.h"
#include "Queue.h"
#include "SysTimer.h"
#include "Message.h"

#include "KeySwap.h"

#include "KeyTest.h"



void main(void)
{
    //系统模块初始化
    Memory_Init( (unsigned char idata *)0xff );        //内部 RAM 的最后端
    Queue_Init();
    Msg_Init(20);                                      //使用 20 字节的消息缓冲区
    SysTimer_Init22M_Default();
```

```
    //通用函数初始化
    KeySwap_Init();

    //按键测试
    KeyTest_Init();

    EA = 1;                                                    //开中断，开始运行系统

    while(1)
    {
        SysTimer_MainLoop();
        Msg_MainLoop();
        KeySwap_MainLoop();

        //按键测试
        KeyTest_MainLoop();
    }
}
```

测试说明：

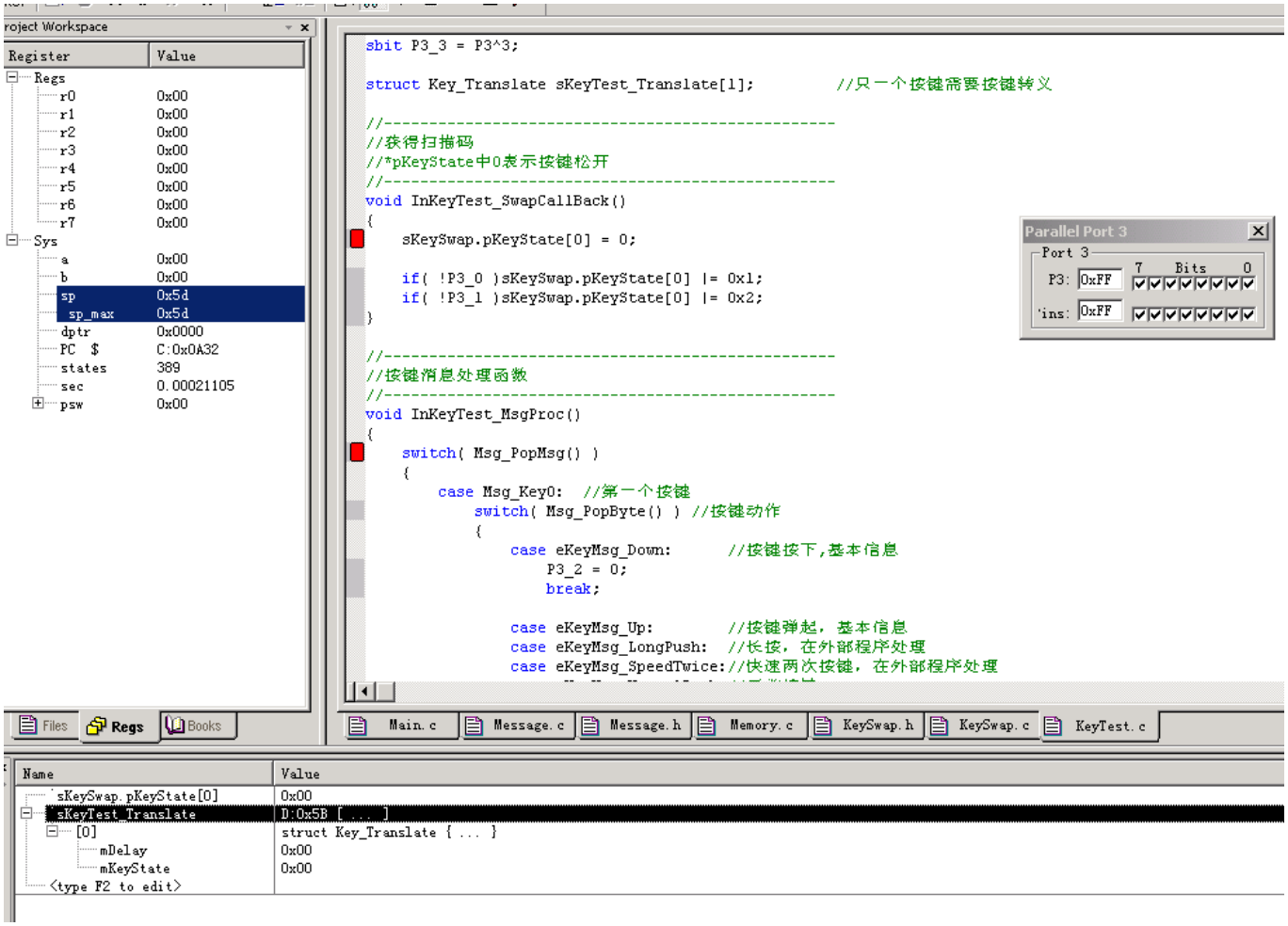
1）在 KeyTest.c 中如下两个位置的设置断点：

```
void InKeyTest_SwapCallBack()
{
    sKeySwap.pKeyState[0] = 0;

    if( !P3_0 )sKeySwap.pKeyState[0] |= 0x1;
    if( !P3_1 )sKeySwap.pKeyState[0] |= 0x2;
}

void InKeyTest_MsgProc()
{
    switch( Msg_PopMsg() )
```

2）在 watch 窗口中添加对 sKeyTest_Translate 的观察。

3）打开 IO 口中 P3 的状态窗口。

界面如下：



3）开始运行程序。每 20MS InKeyTest_SwapCallBack 会被调用一次。在任意一个时间设置 P3.0=0,可以观察到 InKeyTest_MsgProc 被执行，单步运行可以看到 P3.2=0。

4）在任意一个时间设置 P3.1=0,可以观察到 InKeyTest_MsgProc 被执行，单步运行可以看到 KeySwap_Translate 被执行，开始进行按键功能扩展的转义处理。继续运行，在 watch 窗口观察到 sKeyTest_Translate 里面的 mDelay 在减少，当=0 时又会进入 InKeyTest_MsgProc 处的断点，跟踪运行可以看到长按键处理程序 P3_3=0.

5）同样，P3.0、P3.1=1 也有相应消息传递进来，InKeyTest_MsgProc 被执行。

6）继续进行快速两次按键产生消息的测试。在任意一个时间设置 P3.1=0,可以观察到 InKeyTest_MsgProc 被执行，当重新开始进入第一个断点时（InKeyTest_SwapCallBack），设置 P3.1=1，产生按键松开消息。当消息执行完毕再次进入第一个断点断点时设置 P3.1=0，产生快速两次按键效果，这时可以跟踪到再次有按键 DOWN 消息，经过转义后马上插入了 SpeedTwice 消息，可以跟踪到 P3_3 = 1。

程序说明：

KeySwap.h 中包含 enum KeyMsgUnit 枚举定义，为按键消息包含的信息参数定义，作为 Msg_Keyx 的消息参数，表示了按键消息类型。其中 **eKeyMsg_Down**、**eKeyMsg_Up** 为硬件按键产生的基本信息参数，**eKeyMsg_LongPush**、**eKeyMsg_SpeedTwice**、**eKeyMsg_NormalPush** 是需要 **KeySwap_Translate** 按键消息转义对上述两个基本按键消息参数转换得来的。

struct Key_Translate 结构是通用按键扫描模块提供外部模块产生按键消息参数转义用的变量结构，记录了转义过程的状态变化和延时时间。

在有转移发生时（**KeySwap_Translate 被调用**）：

快速两次按键发生，间隔时间<KEYTWICEDELAY 定义的基于 20ms 计数的 0.4 秒时间，则转移函数判定为 eKeyMsg_SpeedTwice。

如按键按下保持时间>KEYLONGDELAYMAX 定义的 1.6 秒，则转移函数判定为 eKeyMsg_LongPush。

其它情况发生，判定为 eKeyMsg_NormalPush。

实际我们只要使用转义发生的 SpeedTwice、LongPush、NormalPush 来处理任务就能满足绝大部分要求，不过按键的基本消息参数 **eKeyMsg_Down**、**eKeyMsg_Up** 同样也可以用来做一般任务处理，只不过不推荐。KeySwap_Translate 函数内部在基本消息参数发生时在不同状态进行切换，其关于计算延时的部分必须以来 KeySwap_TranslateLoop 进行。这两个函数都是要被外部函数包含的。

外部按键控制程序必须提供按键扫描码的回调函数，通用按键扫描模块才能正确发出消息，否则不会有任何关于按键的消息产生。通用按键扫描模块中的结构 struct KeySwap 的 pKeyState 指针指向了按键状态的信息，用 bit 来表示一个按键，0 表示按键释放状态，1 表示按键按下状态。下面是代码实例。

```
void InKeyTest_SwapCallBack()
{
    sKeySwap.pKeyState[0] = 0;

    if( !P3_0 )sKeySwap.pKeyState[0] |= 0x1;
    if( !P3_1 )sKeySwap.pKeyState[0] |= 0x2;
}
```

通用按键扫描模块对 sKeySwap.pKeyState 中每位进行比较判断处理。

外部按键控制程序必须先调用 KeySwap_Register 进行按键信息注册，提供扫描码的回调函数和按键消息的第一个消息字，按键的最大数目。

项目在此状态编译的结果为：

Data = 52.1
Xdata = 0
Code = 2086

8）增加弱实时处理功能

在开始编写通用数码管显示通用程序之前，要先对前面的系统时间管理模块进行一次重大修改，以支持复杂系统下程序的通用性。这个重大修改就是增加弱实时处理功能。

弱实时处理缘由：

程序设计的层次分为三个层次，下述表格表明它们的特征：

	非实时	弱实时	强实时
层次定义	可以将事件的处理延后，直到 MCU 处理能力承受极限	可以放缓处理，但不能超过一个极限时间	必须立刻处理的事件
不能及时处理后果	-	缓冲区可能溢出	错误发生
增加不能及时处理能力的容量	换更高性能的 MCU	增加强实时接口的队列或缓冲容量	换更高性能的 MCU
实现方式	主循环	系统时间 5MS 中断	中断
对应中断等级	-	PT0 = 0	例如 PS=PT1=PX1=PX0=1
使用的 using 寄存器	-	using 1	using 2
可以被嵌套	弱实时中断、强实时中断	强实时中断	-
对象		1）强实时对应的队列或缓冲区。 2）小程序结构可放在主循环，大程序结构放在中断处理的模块。	各种中断
实例	95%-98%的任务或模块	1）对串口队列的数据进行协议处理 2）按键扫描 3）数码管显示驱动 4）其它	精确定时 串口 其它

弱实时处理，可以被简单的认为相当于 WINDOW 编程上的一个线程(通常 W 下的驱动==这里的强实时，对驱动的接口线程==弱实时)，主要目的是把外部信息能及时解析并转化为数据量极小的消息。MCU 上的缓冲区一般都是很有限的，如果对外部信息不及时处理，会造成接口缓冲区的溢出。按键扫描和数码管显示驱动，一样具备弱实时特性：时间要求不苛刻，但又必须在一个时间段内完成。

下面对 SysTimer.c 进行改造，只需很小的改动就可以完成对弱实时处理的支持。下面代码加粗部分为修改部分。

改进后的 SysTimer.h：

```
//-----------------------------------------------------------------------------
//系统时间管理
//-----------------------------------------------------------------------------
#define TOPRELOAD_22M    (144L/4)          //20ms/4
#define TOPRELOAD_11M    (TOPRELOAD_22M/2)

#ifdef _SysTimerH
 //用户可以使用的变量
 bit fSysTimer_Touch;
 bit fSysTimer_FlashTouch;
 bit fSysTimer_FlashState;

#else
```

```
extern bit fSysTimer_Touch;          //系统时间触发
extern bit fSysTimer_FlashTouch;     //闪烁时间触发
extern bit fSysTimer_FlashState;     //闪烁状态

//-------------------------------------------------------------
//构造
//入口：产生 20MS 需要的 T0 高位预装值、闪烁亮的时间计数、闪烁灭的时间计数
//-------------------------------------------------------------
extern void SysTimer_Init( unsigned char mPreLoad, unsigned char mFlashOnCount, unsigned char mFlashOffCount );

//-------------------------------------------------------------
//析构,释放在 init()中使用到的硬件资源
//-------------------------------------------------------------
extern void SysTimer_Destory(void);

//-----------------------------------------------------------------------------
//注册弱实时任务处理函数
//-----------------------------------------------------------------------------
extern void SysTimer_WeeknessRegister( void(*Function_20ms)(), void(*Function_5ms)() );

//-----------------------------------------------------------------------------
//循环
//-----------------------------------------------------------------------------
extern void SysTimer_Loop(void);

//main.c 使用的主循环
#define SysTimer_MainLoop() do{ fSysTimer_Touch = fSysTimer_FlashTouch = 0; SysTimer_Loop(); }while(0)

//22.1184M 系统默认参数配置初始化
#define SysTimer_Init22M_Default() do{ SysTimer_Init(T0PRELOAD_22M, 500/20, 500/20 );}while(0)
//11.0592M 系统默认参数配置初始化
#define SysTimer_Init11M_Default() do{ SysTimer_Init(T0PRELOAD_11M, 500/20, 500/20 );}while(0)

#endif
```

改进后的 SysTimer.c：

```
//-------------------------------------------------------------
//系统时间管理
//T0 产生系统时间 20ms
//T0 中断 5ms，处理弱实时任务。20ms 主要提供按键扫描，5ms 主要提供给数码 LED、协议解析用。
//-------------------------------------------------------------
#define _SysTimerH

#include <reg52.h>
#include "Memory.h"
#include "SysTimer.h"
```

```
bit fInSys_20ms;                        //从中断产生的标志转为主循环系统使用的 20MS 标志需要

//内部使用的变量
struct InSysTimer{
    unsigned char mT0_PreLoad;          //T0 在模式 0 的 13 位模式下的高位初装值,用来产生系统时
间用
    unsigned char mFlashOff_Count;      //闪烁灭的时间计数器
    unsigned char mFlashOn_Count;       //闪烁亮的时间计数器
    unsigned char mFlash_Count;         //闪烁计数

    unsigned char m20msCount;           //从 5MS 产生 20MS 需要的计数器

    void ( *Weekness_20ms )();          //20ms 弱实时任务函数数组指针
    void ( *Weekness_5ms )();           //5ms 弱实时任务函数数组指针
};
struct InSysTimer sInSysTimer;

//---------------------------------------------------------------------
//系统时间初始化
//T0 工作在 16 位的模式 1 下
//入口：产生 20MS 需要的 T0 高位预装值
//---------------------------------------------------------------------
void SysTimer_Init( unsigned char mPreLoad, unsigned char mFlashOnCount, unsigned char
mFlashOffCount )
{
    //变量清 0
    Memory_Memset( (unsigned char *)&sInSysTimer, 0, sizeof(struct InSysTimer) );

    sInSysTimer.mT0_PreLoad = (~mPreLoad) + 1;          //T0 高位预装值
    sInSysTimer.mFlashOn_Count = mFlashOnCount;
    sInSysTimer.mFlash_Count = sInSysTimer.mFlashOff_Count = mFlashOffCount;

    TMOD |= 0x1;                        //T0.M0 = 1;
    TH0 = sInSysTimer.mT0_PreLoad;
    ET0 = 1;
    TR0 = 1;                            //T0 开始运行
}

//---------------------------------------------------------------------
//析构
//---------------------------------------------------------------------
void SysTimer_Destory(void)
{
    TR0 = 0;
    ET0 = 0;
    TMOD &= ~0x1;                       //释放硬件资源
}

//---------------------------------------------------------------------
//注册弱实时任务处理函数
```

```
//-----------------------------------------------------------------------------
void SysTimer_WeeknessRegister( void(*Function_20ms)(), void(*Function_5ms)() )
{
    sInSysTimer.Weekness_20ms = Function_20ms;
    sInSysTimer.Weekness_5ms = Function_5ms;
}


//-----------------------------------------------------------------------------
//循环
//-----------------------------------------------------------------------------
void SysTimer_Loop(void)
{
    if( fInSys_20ms ){
        fInSys_20ms = 0;                //中断中产生的 20MS，不能直接在主循环中使用，必须转换

        fSysTimer_Touch = 1;            //系统时间到达

        //闪烁时间控制
        if( fSysTimer_FlashState ){
            //处于闪烁亮
            if( --sInSysTimer.mFlash_Count == 0 ){
                //进入闪烁灭的状态
                sInSysTimer.mFlash_Count = sInSysTimer.mFlashOff_Count;
                fSysTimer_FlashState = 0;
                fSysTimer_FlashTouch = 1;
            }
        }
        else {
            //处于闪烁灭
            if( --sInSysTimer.mFlash_Count == 0 ){
                //进入闪烁亮的状态
                sInSysTimer.mFlash_Count = sInSysTimer.mFlashOn_Count;
                fSysTimer_FlashState = 1;
                fSysTimer_FlashTouch = 1;
            }
        }
    }
}


//-----------------------------------------------------------------------------
//T0 中断
//处理弱实时任务，并产生 20MS 系统时间
//-----------------------------------------------------------------------------
void Timer0(void) interrupt 1 using 1
{
    TH0 = sInSysTimer.mT0_PreLoad;      //重装 T0 高位初始值

    if( ++sInSysTimer.m20msCount > 3 ){
```

```
            sInSysTimer.m20msCount = 0;
            //20ms 触发
            fInSys_20ms = 1;
            //调用 20ms 中断要处理的弱实时任务
            if( sInSysTimer.Weekness_20ms )(*sInSysTimer.Weekness_20ms)();
        }

        //调用 5ms 中断要处理的弱实时任务。
        if( sInSysTimer.Weekness_5ms )(*sInSysTimer.Weekness_5ms)();
    }
```

程序说明：

修改了系统时间 20ms 产生机制，由原来的查询 TF0 改为中断中产生 fInSys_20ms，用 fInSys_20ms 代替了 TF0。

T0 时间基准由 20MS 改为 5MS。5MS 进行一次弱实时处理，可以在保证最大处理效率。

增加了弱实时任务的回调函数注册。这里我们没有对弱实时任务的个数判断，在 SysTimer.c 中直接接受外部回调函数注，而是只提供固定的一个回调函数，主要是扩展性和效率的一种平衡。目前这种方式同样能支持模块化和标准化，不过需要 main.c 中提供弱实时任务的回调函数集合。这样的特性对比，可以参考 Message.c 中对回调函数注册的处理方式，看它们之间处理方式的不同点。

下面是修改后的 main.c:

```
//-----------------------------------------------------------
//主程序
//-----------------------------------------------------------
#include <reg52.h>

#include "Memory.h"
#include "Queue.h"
#include "SysTimer.h"
#include "Message.h"

#include "KeySwap.h"

#include "KeyTest.h"

//-----------------------------------------------------------
//弱实时处理任务
//20ms 和 5ms
//-----------------------------------------------------------
void Weekness_20msCallback(void)
{
 KeySwap_Loop();
}

void Weekness_5msCallback(void)
{

}
```

```
//-------------------------------------------------------
void main(void)
{
 //系统模块初始化
 Memory_Init( (unsigned char idata *)0xff );        //内部RAM的最后端
 Queue_Init();
 Msg_Init(20, eMSG_End);                            //使用 20 字节的消息缓冲区,最大消息数
(Message.h中)
 SysTimer_Init22M_Default();

 //通用函数初始化
 KeySwap_Init();

 //注册弱实时任务函数
 SysTimer_WeeknessRegister( Weekness_20msCallback, Weekness_5msCallback );

 //按键测试
 KeyTest_Init();

 EA = 1;                                            //开中断，开始运行系统

 while(1)
 {
     SysTimer_MainLoop();
     Msg_MainLoop();

     //按键测试
     KeyTest_MainLoop();
 }
}
```

KeySwap.h中的主循环中的宏同样做修改，将原来20MS一次进行查询改为20ms一次中断：删除KeySwap_MainLoop()宏定义。

9）通用数码 LED 显示模块

这个模块封装了对 8 段数码 LED 管的通用处理部分，和通用按键扫描模块一样，将输出显示和硬件有关部分采用回调函数方式由外部提供。现在这个模块只对最大 8 个数码管进行管理，采用最常用的 COM 控制，SEG 输出位码方式。

该模块有时间限制要求，归于弱实时处理部分。这里将在系统提供的 5MS 弱实时中断中被调用，这样也可以保证显示刷新率为：1000/(5*8)=50Hz 刷新率，远大于人眼观察不抖动需要的 24Hz。

该模块还封装了一个默认的显示输出转换码表，该码表也可以被外部的码表替换，以适应不同场合的硬件输出，充分保证了设计的灵活性。

DigitTube.h 文件：

```
//-------------------------------------------------------
//数码管显示通用模块
//默认使用的数码表：
//默认显示顺序为：0、1、2、3、4、5、6、7、8、9、A、b、C、d、E、F、8 个单独位、P、t、L、H、特殊符号。
//
//数码管转换表可以被外部替换。
//默认显示转换表：
//   0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,   (0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,)
//   0x7f,0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71,   (0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,)
//   0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,   (0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,)
//   0x73,0x78,0x38,0x76,0x43,0x4C,0x61,0x58,   (0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F,)
//   0x41,0x48,0x03,0x0C,0x18,0x21                  (0x20,0x21,0x22,0x23,0x24,0x25)
//最高位=1 显示小数点。
//-------------------------------------------------------

struct DigitTube{
    unsigned char aDispBuffer[8];                  //显示缓冲区开始指针
};

#ifdef _DigitTubeH
    //用户使用的变量
    struct DigitTube sDigitTube;

#else
    extern struct DigitTube sDigitTube;
    extern unsigned char code cDigitTable[];       //系统默认的显示码表

    //-------------------------------------------------------------
    //初始化
    //-------------------------------------------------------------
    extern void DigitTube_Init(void);

    //-------------------------------------------------------------
    //析构
    //-------------------------------------------------------------
    extern void DigitTube_Destory(void);

    //-------------------------------------------------------------
    //注册回调函数
    //-------------------------------------------------------------
```

```
        extern void DigitTube_CallBackRegister( void (*CallBack_DigitTube)(unsigned char mComData, unsigned char mSegData) );

        //-------------------------------------------------------------
        //循环,5ms 调入一次。
        //-------------------------------------------------------------
        extern void DigitTube_Loop(void);

    #endif
```

DigitTube.c 文件：

```
    //-------------------------------------------------------------
    //数码管显示通用模块
    //最大 8 个数码管
    //外部还负责提供两个外部硬件函数，分别为：
    //  DigitTube_Callback(unsigned char mComData, unsigned char mSegData)，输出段选数据、位选数据
    //-------------------------------------------------------------
    #define _DigitTubeH

    #include <reg52.h>
    #include "Memory.h"

    #include "DigitTube.h"

    #define a 0x01
    #define b 0x02
    #define c 0x04
    #define d 0x08
    #define e 0x10
    #define f 0x20
    #define g 0x40
    #define h 0x80

    //默认使用的数码表
    //默认显示顺序为：0、1、2、3、4、5、6、7、8、9、8 个单独位、A、b、C、d、E、F、P、t、L、H、特殊符号。
    //最高位=1 带小数点显示。其它数据消隐（空白）。
    unsigned char code cDigitTable[]={
        a|b|c|d|e|f,        //0
        b|c,                //1
        a|b|d|e|g,          //2
        a|b|c|d|g,          //3
        b|c|f|g,            //4
        a|c|d|f|g,          //5
        a|c|d|e|f|g,        //6
        a|b|c,              //7
        a|b|c|d|e|f|g,      //8
        a|b|c|d|f|g,        //9
        a|b|c|e|f|g,        //A
        c|d|e|f|g,          //b
        a|d|e|f,            //C
```

```
    b|c|d|e|g,          //d
    a|d|e|f|g,          //E
    a|e|f|g,            //F

    //0x10 开始
    a,
    b,
    c,
    d,
    e,
    f,
    g,
    h,
    a|b|e|f|g,          //P
    d|e|f|g,            //t
    d|e|f,              //L
    b|c|e|f|g,          //H

    //特殊符号
    a|b|g,
    c|d|g,
    a|f|g,
    d|e|g,
    a|g,
    d|g,
    a|b,
    c|d,
    d|e,
    a|f
};

//数字 ID 转换为逻辑位用的表
unsigned char code cIndexToLogicBit[] = { 0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80 };

struct InDigitTube{
    void (*CallBack_DigitTube)(unsigned char mComData, unsigned char mSegData);    //显示输出数据回调函数指针。

    unsigned char mComId;                    //输出 COM 的索引
};
struct InDigitTube sInDigitTube;

//------------------------------------------------------------
//初始化
//入口：>1 外部显示转换表，=0 内部默认转换表，=1 缓冲数据直接输出
//------------------------------------------------------------
void DigitTube_Init(void)
{
    Memory_Memset( (unsigned char *)&sInDigitTube, 0, sizeof(struct InDigitTube) );
    Memory_Memset( (unsigned char *)&sDigitTube, 0, sizeof(struct DigitTube) );
```

```
}

//------------------------------------------------------------
//析构
//------------------------------------------------------------
void DigitTube_Destory(void){}

//------------------------------------------------------------
//注册回调函数
//------------------------------------------------------------
void DigitTube_CallBackRegister( void (*CallBack_DigitTube)(unsigned char mComData, unsigned char mSegData) )
{
    sInDigitTube.CallBack_DigitTube = CallBack_DigitTube;
}

//------------------------------------------------------------
//循环
//5ms 调入一次。弱实时中断调用
//------------------------------------------------------------
#pragma NOAREGS
void DigitTube_Loop(void)
{
    //硬件输出 COM 数据和 SEG 数据
    if( sInDigitTube.CallBack_DigitTube ){
        (*sInDigitTube.CallBack_DigitTube)(                cIndexToLogicBit[sInDigitTube.mComId], sDigitTube.aDispBuffer[ sInDigitTube.mComId ] );
    }

    //COM 选择移位
    if( ++sInDigitTube.mComId >7 ){
        sInDigitTube.mComId = 0;
    }
}
#pragma AREGS
```

程序说明：

1） DigitTube.h 文件中 struct DigitTube sDigitTube 是提供给外部使用的显示缓冲区，对该缓冲区的写入会改变数码管显示结果。

2） void DigitTube_CallBackRegister(void (*CallBack_DigitTube)(unsigned char mComData, unsigned char mSegData))是外部提供该通用数码 LED 显示模块的硬件回调函数，CallBack_DigitTube 为函数指针，unsigned char mComData, unsigned char mSegData 则表示该函数具有两个入口参数。

3） DigitTube_Loop()是系统弱实时 5MS 中断要调用的函数：5MS 进行一次一个位的扫描显示。因为弱实时中断使用了 using 1 寄存器区，所以对该函数的编译，必须加上#pragma NOAREGS 开关，避免产生寄存器区切换的状态下发生对 R 寄存器的直接地址寻址的指令（如：ANL 0x5, x，加上该开关则为正确的 ANL R5, x）。

4） 该模块同样可以生成标准库。

10） 驱动程序设计

驱动程序的概念：

实现外部硬件的时序和硬件的缓冲机制，并按一定的接口模式提供外部非实时或弱实时程序处理用。

驱动程序的特点：

和硬件相关，在不同的应用场合会有相应的配置变化。如串口中断的等级变化、8 位数据位和 9 位数据位的选择、按键码扫描的硬件和逻辑变化等等。如果只使用在某些特定场合，则可以生成相应的库。如果保留通用性，最好按 C 原文件提供。

下面开始三个驱动程序设计：串口驱动、按键驱动、数码管显示驱动。其中按键驱动调用按键通用模块，数码管显示驱动调用通用数码 LED 显示模块，实现真正意义上能被调用的驱动程序。

串口驱动模块：

该模块的串口接收和发送使用通用队列作缓冲。接收缓冲区的大小，直接决定了缓冲时间：缓冲区越大，留给主程序处理的时间也就越长，支持的程序规模也就越大，但消耗的内存也就越大。所以选择一个合适的缓冲区大小，是程序员要慎重考虑的。

采用队列作缓冲后，只需对接收队列进行判断和处理就可以有充分的时间完成串口数据的解析，同样对发送队列的写入数据，可以自动地引起串口数据的发送。

下面的串口驱动模块，只完成 8 位数据模式下的处理，没有对 9 位数据（有奇校验或偶校验）处理。

Uart08.h 文件：

```
//-----------------------------------------------------------
//UART0 驱动
//波特率转换定时器：TimerLoad = 256-2^n*f/384/Baud。(n=加倍，f=MCU 工作频率，Baud=波特率)
//UART0 总是使用 T1
//n 总是为 1
//-----------------------------------------------------------

#define Baud_1200_11M        48
#define Baud_2400_11M        24
#define Baud_4800_11M        12
#define Baud_9600_11M        6
#define Baud_19200_11M       3
#define Baud_38400_11M       2
#define Baud_57600_11M       1

#define Baud_1200_22M        96
#define Baud_2400_22M        48
#define Baud_4800_22M        24
#define Baud_9600_22M        12
#define Baud_19200_22M       6
#define Baud_38400_22M       3
#define Baud_57600_22M       2
#define Baud_115200_22M      1


//-----------------------------------------------------------
struct Uart0{
 struct QueueBuffer idata *psSend;              //发送队列指针
 struct QueueBuffer idata *psReci;              //接收队列指针
};

#ifdef _UART08H
 bit fUart0_Send_Ok;
 //用户使用的变量
 data struct Uart0 sUart0;
#else
 extern bit fUart0_Send_Ok;
 extern data struct Uart0 sUart0;

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

```
    //初始化
    //和通用队列连接。串口方式为无校验位，总是使用 T1
    //入口：发送队列大小、接收队列大小、波特率
    //-------------------------------------------------------------
    extern void Uart08_Init(unsigned char mSendBufferMax, unsigned char mReciBufferMax, unsigned char
mBaud);

    //-------------------------------------------------------------
    //析构
    //-------------------------------------------------------------
    extern void Uart08_Destory(void);

    #define                                                                    Uart08_Push(x)
 do{Queue_Push(sUart0.psSend,x);if(!fUart0_Send_Ok){fUart0_Send_Ok=1;TI=1;};}while(0)
    #define Uart08_Pop()        Queue_Pop(sUart0.psReci)
    #define Uart08_Read(x)      Queue_Read(sUart0.psReci,x)
    #define Uart08_ReciNum()    Queue_Num(sUart0.psReci)
    #define Uart08_SendClear()  do{ Queue_Clear(sUart0.psSend);}while(0)
    #define Uart08_ReciClear()  do{ Queue_Clear(sUart0.psReci);}while(0)

#endif
```

Uart08.c 文件：

```
//-------------------------------------------------------------
//UART0 驱动
//-------------------------------------------------------------
#define _UART08H

#include <reg52.h>

#include "Memory.h"
#include "Queue.h"

#include "Uart08.h"

//-------------------------------------------------------------
//初始化
//和通用队列连接。串口方式为无校验位，总是使用 T1
//入口：发送队列大小、接收队列大小、波特率
//-------------------------------------------------------------
void Uart08_Init(unsigned char mSendBufferMax, unsigned char mReciBufferMax, unsigned char mBaud)
{
 fUart0_Send_Ok = 0;
 Memory_Memset( (unsigned char *)&sUart0, 0, sizeof(struct Uart0) );

 sUart0.psSend = Queue_Register( mSendBufferMax );          //申请发送队列
 sUart0.psReci = Queue_Register( mReciBufferMax );          //申请接收队列

 //设置串口模式
 PCON |= 0x80;                    //SMOD1 = 1，波特率加倍
```

```
 TMOD = (TMOD&0x0f) | 0x20;        //自动重装
 TL1 = TH1 = (~mBaud)+1;           //T1 周期为波特率周期
 TR1 = 1;                          //T1 运行
 //串口方式:无校验
 SCON = 0x40|0x10;                 //SM1 = 1;REN = 1;
 ES = 1;                           //串口中断打开
}

//-----------------------------------------------------------
//析构
//-----------------------------------------------------------
void Uart08_Destory(void)
{
 TR1 = 0;
 TMOD &= 0xf;
 PCON &= ~0x80;

 ES = 0;
 SCON = 0;
}

//-----------------------------------------------------------
//串口 0 中断
//使用一般等级中断
//-----------------------------------------------------------
void UART08(void) interrupt 4 using 1
{
 if( TI ){
      TI = 0;
      if( Queue_Num(sUart0.psSend) ){
           SBUF = Queue_Pop( sUart0.psSend );    //发送
      }
      else {
           fUart0_Send_Ok = 0;                          //发送队列空，不再发送
      }
 }

 if( RI ){
      RI = 0;
      Queue_Push( sUart0.psReci, SBUF );              //接收数据存入队列。
 }
}
```

程序设计说明：

1） MCU 的工作频率为能生成标准波特率的 22.1184M 和 11.0592M。

2） 串口 0 总是使用 T1。因为 T2 的功能远比 T1 强大，所以要保留给其它模块使用。

3） 总是使用了波特率加倍标志，这样波特率宏定义简单。

4） 检测串口数据，只需检测串口接收队列状况。

5） 发送串口数据，只需对串口队列写入。

6） 发送和接收串口数据，都使用了中断方式。

7） 串口中断等级选择：

小程序下,可以选择和弱实时处理中断等级相同的低等级中断,可以减少一个寄存区的使用：使用 using 1。在大程序结构下，必须使用高等级中断级别：设置 PS = 1,同时使用 using 2，这样可保证接收中断能及时响应并存入到缓冲区。

测试程序：

串口测试程序非常简单，这次直接在 main.c 中添加测试代码：

20ms 人工触发 RI，模拟串口接收，并累计到 3 个数据时串口发送出去。

Main.c 文件：

```
//-------------------------------------------------------------
//主程序
//-------------------------------------------------------------
#include <reg52.h>

//系统模块
#include "Memory.h"
#include "Queue.h"
#include "SysTimer.h"
#include "Message.h"

//通用模块
#include "KeySwap.h"
#include "DigitTube.h"

//驱动模块
#include "Uart08.h"

//测试
#include "KeyTest.h"
#include "DigitTest.h"

//-------------------------------------------------------------
//弱实时处理任务
//20ms 和 5ms
//-------------------------------------------------------------
void Weekness_20msCallback(void)
{
 //按键扫描
 KeySwap_Loop();
}

void Weekness_5msCallback(void)
{
 //数码 LED 通用驱动
 DigitTube_Loop();
}
```

```
//-------------------------------------------------------------
void main(void)
{
 //系统模块初始化
 Memory_Init( (unsigned char idata *)0xff );    //内部 RAM 的最后端
 Queue_Init();
 Msg_Init(20, eMSG_End);                        //使用 20 字节的消息缓冲区,最大消息数(Message.h 中)
 SysTimer_Init22M_Default();

 //注册弱实时任务函数
 SysTimer_WeeknessRegister( Weekness_20msCallback, Weekness_5msCallback );

 //通用函数
 KeySwap_Init();
 DigitTube_Init(0);                             //使用内部显示转换表

 //驱动
 Uart08_Init( 10, 10, Baud_9600_22M );          //发送队列和接收队列使用 10 字节.9600bps

 //测试
 KeyTest_Init();
 DigitTest_Init();

 EA = 1;                                        //开中断，开始运行系统

 while(1)
 {
     SysTimer_MainLoop();
     Msg_MainLoop();

     //按键测试
     KeyTest_MainLoop();
     //数码管显示测试
     DigitTest_MainLoop();

     //Uart0 驱动测试
     if( fSysTimer_Touch ){
         //20ms 人工触发 RI，模拟接收，并累计到 3 个数据时发送出去
         RI = 1;
         if( Uart08_ReciNum()>2 ){
             //串口有数据接收
             while( Uart08_ReciNum() ){
                 Uart08_Push( Uart08_Pop() );//将接收到的数据从队列中弹出，并压入发送缓冲发送。
             }
         }
     }

 }
}
```

数码管 LED 驱动模块：

DigitDrv.h 文件：

```
//-------------------------------------------------
//初始化，注册显示输出的硬件部分的回调函数
//-------------------------------------------------
extern void DigitDrv_Init(void);

//-------------------------------------------------
//析构
//-------------------------------------------------
extern void DigitDrv_Destory(void);

//读指定位置的显示数据
#define DigitDrv_Read(mLocation) (sDigitTube.aDispBuffer[mLocation])

//直接写显示数据
#define DigitDrv_Write(mLocation, mDispData)   do{sDigitTube.aDispBuffer[mLocation]                =
mDispData;}while(0)

//通过默认码表写显示数据
#define DigitDrv_WriteTable(mLocation, mDispData)  do{sDigitTube.aDispBuffer[mLocation]           =
cDigitTable[mDispData];}while(0)
```

DigitDrv.c 文件：

```
//-------------------------------------------------------------
//数码 LED 显示驱动
//调用通用数码显示模块
//使用数据直接输出显示方式，支持单个 LED*8 的显示功能
//-------------------------------------------------------------
#include <reg52.h>

#include "Memory.h"
#include "DigitTube.h"

//-------------------------------------------------
//通用数码 LED 输出硬件需要的回调函数。
//入口参数：COM 数据、SEG 数据
//COM 数据中 0 表示无输出，和原理图事故是反的，过 P0 输出时要对此数据取反。
//-------------------------------------------------
#pragma NOAREGS
void InDigitDrv_OutCallBack(unsigned char mComData, unsigned char mSegData)
{
    P2 = 0;                                    //关闭显示，避免抖动
    P0 = ~mComData;
    P2 = mSegData;
}
#pragma AREGS

//-------------------------------------------------
//初始化，注册显示输出的硬件部分的回调函数
```

```
//-------------------------------------------------
void DigitDrv_Init(void)
{
    DigitTube_CallBackRegister( InDigitDrv_OutCallBack );
}

//-------------------------------------------------
//析构
//-------------------------------------------------
void DigitDrv_Destory(void)
{
    P0 = 0xff;
    P2 = 0xff;
}
```

程序说明：

DigitDrv 模块比较简单，提供了 DigitTube 通用模块的硬件回调函数，并在 h 文件中采用宏定义的方式，向外部程序提供了两种方式写显示缓冲区：DigitDrv_Write(mLocation, mDispData) 和 DigitDrv_WriteTable(mLocation, mDispData)。前一个宏是直接将数据写入指定显示位置，后一个函数是通过默认的显示转换表将数据转换后写入。这样即可支持标准的 8 段码显示，也可支持任意数据的显示。后面这点在用 LED 阵列构成数码管显示的方式下特别重要，参看原理图。

按键驱动程序：

和前面的 DigitDrv 模块一样，提供按键通用模块的硬件扫描码回调函数，并根据实际需要提供了按键消息的转换：将按键物理消息转换为其它模块要使用的任务消息。

KetDrv.h 文件：

```
//初始化
//-------------------------------------------------
extern void KeyDrv_Init();

//-------------------------------------------------
//析构
//-------------------------------------------------
extern void KeyDrv_Destory();

//-------------------------------------------------
//循环
//20ms 执行一次
//-------------------------------------------------
extern void KeyDrv_Loop();

#define KeyDrv_MainLoop()   do{ if(fSysTimer_Touch)KeyDrv_Loop(); }while(0)
```

KeyDrv.c 文件

```
//-----------------------------------------------------------
//按键驱动
//采用消息传递给其它模块
//-----------------------------------------------------------
#include <reg52.h>

#include "Memory.h"
#include "Message.h"

#include "KeySwap.h"

struct Key_Translate sKeyDrv_Translate[2];                  //只 K5、K6 按键需要按键转义

sbit P1_0=P1^0;
//-------------------------------------------------
//获得扫描码
//*pKeyState 中 0 表示按键松开
//-------------------------------------------------
#pragma NOAREGS
void InKeyDrv_SwapCallBack(void)
{
    register unsigned char i,j;

    sKeySwap.pKeyState[0] = 0;

    i = P2;                                                 //保存 P2 输出状态
    j = P0;                                                 //保存 P0 输出状态
    P0 = 0xff;                                              //显示关闭，避免闪烁
```

```
    P2 = 0xff;                                          //设置 P2 为输入状态

    P1_0 = 0;
    sKeySwap.pKeyState[0] = P2;
    P1_0 = 1;

    P2 = i;                                             //恢复 P2 输出
    P0 = j;
}
#pragma AREGS

//-------------------------------------------------
//按键消息处理函数
//-------------------------------------------------
void InKeyDrv_MsgProc(void)
{
    unsigned char mMsg;

    mMsg = Msg_PopMsg();
    switch( mMsg )
    {
        case Msg_Key0:     //K1
        case Msg_Key1:     //K2
        case Msg_Key2:     //K3
        case Msg_Key3:     //K4
            switch( Msg_PopByte() )     //按键动作
            {
                case eKeyMsg_Down:      //按键按下,基本信息
                    //发出转为任务消息
                    Msg_Push( mMsg - Msg_Key0 + Msg_Key_Setup );
                    break;

                case eKeyMsg_Up:        //按键弹起，基本信息
                case eKeyMsg_LongPush: //长按，在外部程序处理
                case eKeyMsg_SpeedTwice://快速两次按键，在外部程序处理
                case eKeyMsg_NormalPush://正常按键
                    break;
            }
            break;

        case Msg_Key4:     //K5
            switch( Msg_PopByte() )
            {
                case eKeyMsg_Down:
                case eKeyMsg_Up:
                    //开始按键消息转义,只对 eKeyMsg_Down、eKeyMsg_Up 这两个基本按键消息起作用
                    KeySwap_Translate( &sKeyDrv_Translate[0], Msg_Key4, Msg_PopByte() );
                    break;

                case eKeyMsg_LongPush:
```

```
            case eKeyMsg_NormalPush:
                break;

            case eKeyMsg_SpeedTwice://快速两次按键，在外部程序处理
                //发出转为任务消息
                Msg_Push( Msg_Key_D5CPL );
                break;
        }
        break;

    case Msg_Key5:     //K6
        switch( Msg_PopByte() )
        {
            case eKeyMsg_Down:
            case eKeyMsg_Up:
                //开始按键消息转义,只对 eKeyMsg_Down、eKeyMsg_Up 这两个基本按键消息起作用
                KeySwap_Translate( &sKeyDrv_Translate[1], Msg_Key5, Msg_PopByte() );
                break;

            case eKeyMsg_LongPush: //长按，在外部程序处理
                //发出转为任务消息
                Msg_Push( Msg_Key_D6CPL );
                break;

            case eKeyMsg_SpeedTwice:
            case eKeyMsg_NormalPush:
                break;
        }
        break;

    case Msg_Key6:     //K7
        switch( Msg_PopByte() )
        {
            case eKeyMsg_Down:
                //发出转为任务消息
                Msg_Push( Msg_Key_D7CPL );
                break;

            case eKeyMsg_Up:
            case eKeyMsg_LongPush:
            case eKeyMsg_SpeedTwice:
            case eKeyMsg_NormalPush:
                break;
        }
        break;

    case Msg_Key7:     //K8
        switch( Msg_PopByte() )
        {
            case eKeyMsg_Down:
```

```
                    break;

                case eKeyMsg_Up:
                    //发出转为任务消息
                    Msg_Push( Msg_Key_D8CPL );
                    break;

                case eKeyMsg_LongPush:
                case eKeyMsg_SpeedTwice:
                case eKeyMsg_NormalPush:
                    break;
            }
            break;
        }
    }

    //-------------------------------------------------
    //初始化
    //-------------------------------------------------
    void KeyDrv_Init(void)
    {
        Memory_Memset( (unsigned char *)&sKeyDrv_Translate[0], 0, sizeof(struct Key_Translate) );
        //注册按键扫描管理,8 个按键
        KeySwap_Register( InKeyDrv_SwapCallBack, Msg_Key0, 8 );

        //注册按键消息
        Msg_Register( InKeyDrv_MsgProc, Msg_Key0, Msg_Key7 );//KeyDrv_MsgProc()处理 Msg_Key0 到 Msg_Key7
之间消息
    }

    //-------------------------------------------------
    //析构
    //-------------------------------------------------
    void KeyDrv_Destory(void){ }

    //-------------------------------------------------
    //循环
    //20ms 执行一次
    //-------------------------------------------------
    void KeyDrv_Loop(void)
    {
        KeySwap_TranslateLoop( &sKeyDrv_Translate[0], Msg_Key4 );            //K5
        KeySwap_TranslateLoop( &sKeyDrv_Translate[1], Msg_Key5 );            //K6
    }
```

11） **按键功能处理任务模块：**

这里开始要处理的事务相关的任务模块的设计。

按键功能处理任务模块的实现很简单：只要处理 KeyDrv.c 模块传递过来的按键消息进行相应的处理就可以了（K5-K8）。

DemoTask.h 文件：

```
//-------------------------------------------------
//初始化
//-------------------------------------------------
extern void DemoTask_Init(void);

//-------------------------------------------------
//析构
//-------------------------------------------------
extern void DemoTask_Destory(void);
```

DemoTask.c 文件：

```
//-----------------------------------------------------------
//按键对应的任务处理任务演示
//对应 Msg_Key_D5CPL 到 Msg_Key_D8CPL 之间的消息
//-----------------------------------------------------------
#include <reg52.h>

#include "Memory.h"
#include "SysTimer.h"
#include "Message.h"

#include "DigitTube.h"
#include "DigitDrv.h"

//-------------------------------------------------
//按键消息处理函数
//com5 为 LED 阵列
//-------------------------------------------------
void InDemoTask_MsgProc(void)
{
    unsigned char i;

    i = DigitDrv_Read(5);

    switch( Msg_PopMsg() )
    {
        case Msg_Key_D5CPL:     //快速连续两次按键时翻转 D5 状态
            i ^= 0x10;
            break;

        case Msg_Key_D6CPL:     //长按键翻转 D6 状态
            i ^= 0x20;
            break;
```

```
            case Msg_Key_D7CPL:     //在按下时翻转 D7 状态
                i ^= 0x40;
                break;

            case Msg_Key_D8CPL:     //在松开时翻转 D8 状态
                i ^= 0x80;
                break;
        }
        DigitDrv_Write( 5, i );
    }

    //-------------------------------------------------
    //初始化
    //-------------------------------------------------
    void DemoTask_Init(void)
    {
        //注册按键消息。KeyTest_MsgProc()处理 Msg_Key_D5CPL 到 Msg_Key_D8CPL 之间消息
        Msg_Register( InDemoTask_MsgProc, Msg_Key_D5CPL, Msg_Key_D8CPL );

    }

    //-------------------------------------------------
    //析构
    //-------------------------------------------------
    void DemoTask_Destory(void){ }
```

程序设计说明：

DemoTask 模块处理消息：Msg_Key_D5CPL、Msg_Key_D6CPL、Msg_Key_D7CPL、Msg_Key_D8CPL 四个任务消息,分别对相应的 LED 进行状态取反。这些 LED 位置在显示缓冲区的 sDigitTube.aDispBuffer[5]，对应于 COM5。

这时的 User_Message.txt 中包含的消息内容：

```
    //按键基本消息
    Msg_Key0,
    Msg_Key1,
    Msg_Key2,
    Msg_Key3,
    Msg_Key4,
    Msg_Key5,
    Msg_Key6,
    Msg_Key7,

    //按键任务处理消息
    Msg_Key_Setup,          //设置开关键，用于开始进行时间设置和退出
    Msg_Key_Up,             //向上减小键
    Msg_Key_Down,           //向下增大键
    Msg_Key_Return,         //确认键
    Msg_Key_D5CPL,          //D5 显示翻转。K5
    Msg_Key_D6CPL,          //D6 显示翻转。K6
    Msg_Key_D7CPL,          //D5 显示翻转。K7
    Msg_Key_D8CPL,          //D6 显示翻转。K8
```

测试用例：

设计到目前为止，已经完成了 90%的设计任务，只余和时间有关的控制模块。这时有必要进行一次比较完整的测试。下面的测试程序就是要完成这点：模拟了按键产生时序而产生任务需要的消息，这样可以很方便地进行软件模拟测试。

SimulantKey.h 文件：

```
//-----------------------------------------------------------
//SimulantKey.c
//按照设定的时间表格产生按键动作
//-----------------------------------------------------------

//-----------------------------------------------------------
//初始化
//-----------------------------------------------------------
extern void SimulantKey_Init(void);

//-----------------------------------------------------------
//主循环
//20ms 进入一次
//-----------------------------------------------------------
extern void SimulantKey_Loop(void);

#define SimulantKey_Loop()  do{ if(fSysTimer_Touch)SimulantKey_Loop();}while(0)
```

SimulantKey.c 文件：

```
//-----------------------------------------------------------
//SimulantKey.c
//按照设定的时间表格产生按键动作
//-----------------------------------------------------------
#include <reg52.h>

#include "Message.h"
#include "KeySwap.h"

sbit P1_0 = P1^0;

//20ms 按键变化一次的状态表及索引
unsigned char idata mKeyId;
unsigned char idata mKeyCount;

//按键扫描码，20MS 计数值
code unsigned char cKeyState[]={
    0x00,    2,                         //无按键动作

    //K5 快速两次按键
    0x10,    6,                         //K5 按下动作
    0x00,    10,                        //K5 释放
    0x10,    6,                         //K5 按下动作
    0x00,    6,                         //K5 释放
```

```
//K6 长按键
0x20,    90,
0x00,     6,

//K7,检测按下功能
0x40,    6,
0x00,     6,

//K8，检测松开功能
0x80,    6,
0x00,     6,

//K1 设置键
0x01,     6,
0x00,     6,

//K2Up 键,加 10。小时位。
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,
0x02,     6,
0x00,     6,

//K3Down 键，减 1
0x04,     6,
0x00,     6,

//K4Return
0x08,     6,
0x00,     6,

//K2Up 键,加 2。分钟位
0x02,     6,
0x00,     6,
```

```
    0x02,    6,
    0x00,    6,

    //K3Down 键，减 1
    0x04,    6,
    0x00,    6,

    //K4Return
    0x08,    6,
    0x00,    6,

    //K2Up 键,加 1。秒位
    0x02,    6,
    0x00,    6,

    //K3Down 键，减 3
    0x04,    6,
    0x00,    6,
    0x04,    6,
    0x00,    6,
    0x04,    6,
    0x00,    6,

    //K4Return
    0x08,    6,
    0x00,    6,

};

//模拟按键的发生用。代替 KeyDrv.c 中的按键扫描回调函数
#pragma NOAREGS
void InDemoKey_SwapCallBack(void)
{
    register unsigned char i,j;

    sKeySwap.pKeyState[0] = 0;

    i = P2;                                          //保存 P2 输出状态
    j = P0;                                          //保存 P0 输出状态
    P0 = 0xff;                                       //显示关闭，避免闪烁
    P2 = 0xff;                                       //设置 P2 为输入状态

    P1_0 = 0;
    sKeySwap.pKeyState[0] = cKeyState[mKeyId];
    P1_0 = 1;

    P2 = i;                                          //恢复 P2 输出
    P0 = j;
}
#pragma AREGS
```

```
//-----------------------------------------------------------
//初始化
//-----------------------------------------------------------
void SimulantKey_Init(void)
{
    //模拟按键的发生用。代替 KeyDrv.c 中的按键扫描回调函数
    KeySwap_Register( InDemoKey_SwapCallBack, Msg_Key0, 8 );

    mKeyId = 0;
    mKeyCount = cKeyState[1];
}

//-----------------------------------------------------------
//析构
//-----------------------------------------------------------
//void SimulantKey_Destory(void){}

//-----------------------------------------------------------
//主循环
//20ms 进入一次
//-----------------------------------------------------------
void SimulantKey_Loop(void)
{
    if( --mKeyCount == 0 ){
        if( mKeyId == 0x50 )
        {
            mKeyCount = 0x50;
        }

        if( mKeyId == sizeof(cKeyState)-2 )
        {
            mKeyId = 0;
        }
        else {
            mKeyId+=2;
        }
        mKeyCount = cKeyState[mKeyId+1];
    }
}
```

程序说明：

cKeyState[]数组存放按键模拟扫描码和延时时间。mKeyId 为索引指针，每次变化为 2。mKeyCount 为延时时间计数器，每 20MS 减 1，变化到 0 会引起 mKeyId 索引指针加 2，重新取数。

调用了 KeySwap_Register(InDemoKey_SwapCallBack, Msg_Key0, 8)函数，覆盖 KeyDrv.c 中注册的函数，用 cKeyState[]表格中含的信息产生按键的模拟信息。

测试说明：

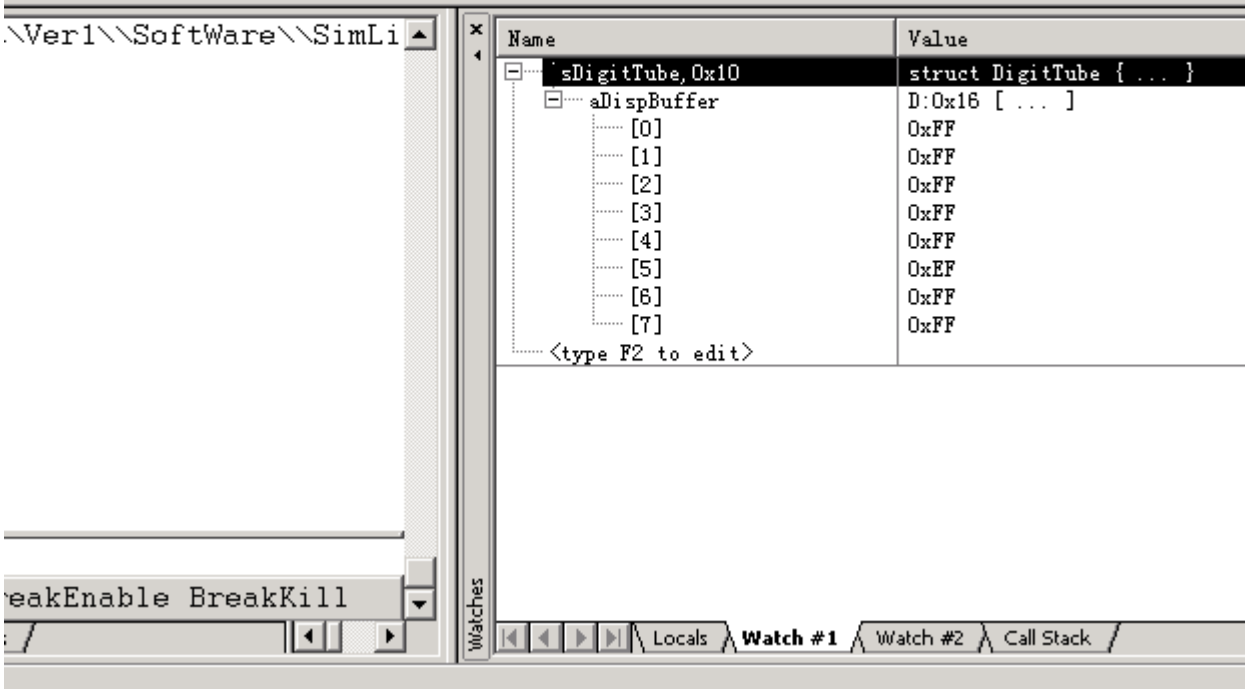
1） 在 Watch 窗口中添加 sDigitTube 的观察，并展开。

2） 在 DemoTask.c 中的消息处理函数 InDemoTask_MsgProc 的四个消息处理部分添加断点。界面如

下：

```
#include "Universal.h"

#include "DigitDrv.h"

//-----------------------------------------------------
//按键消息处理函数
//com5为LED阵列
//-----------------------------------------------------
void InDemoTask_MsgProc(void)
{
    unsigned char i;

    i = DigitDrv_Read(5);

    switch( Msg_PopMsg() )
    {
        case Msg_Key_D5CPL: //快速连续两次按键时翻转D5状态
            i ^= 0x10;
            break;

        case Msg_Key_D6CPL: //长按键翻转D6状态
            i ^= 0x20;
            break;

        case Msg_Key_D7CPL: //在按下时翻转D7状态
            i ^= 0x40;
            break;

        case Msg_Key_D8CPL: //在松开时翻转D8状态
            i ^= 0x80;
            break;
    }
    DigitDrv_Write( 5, i );
```

Main.c | DEMOTASK.C



3） 连续运行，可以观察到断点停留是根据 eyState[]数组中的信息产生的，而且间隔时间也是不同的。同时可以观察到数码管显示输出缓冲区的[5]位在变化。

12） 算法模块

时间控制模块和时间设置模块中涉及了 BCD 加法和 BCD 减法算法，做这两模块前，先完成要用到的 BCD 算法程序。

Math.h 文件：

```
//-------------------------------------------------------------
//BCD 加 1
//入口：BCD 数据,限制的最大数，限制的最小数
//出口：BCD+1 后数据。如果==限制的最小数则表示发生溢出。
//-------------------------------------------------------------
extern unsigned char Math_BcdInc( unsigned char mBCD, unsigned char mLimitMax, unsigned char mLimitMin );

//-------------------------------------------------------------
//BCD 加 1
//入口：BCD 数据,限制的最大数，限制的最小数
//出口：BCD-1 后数据。如果==限制的最大数则表示发生溢出。
//-------------------------------------------------------------
extern unsigned char Math_BcdDec( unsigned char mBCD, unsigned char mLimitMax, unsigned char mLimitMin );
```

Math.c 文件：

```
//-------------------------------------------------------------
//算法模块
//-------------------------------------------------------------

//-------------------------------------------------------------
//BCD 加 1
//入口：BCD 数据,限制的最大数，限制的最小数
//出口：BCD+1 后数据。如果==限制的最小数则表示发生溢出。
//-------------------------------------------------------------
unsigned char Math_BcdInc( unsigned char mBCD, unsigned char mLimitMax, unsigned char mLimitMin )
{
     if( mBCD == mLimitMax ){
          return mLimitMin;
    }

     mBCD++;
     if( (mBCD&0x0f) > 9 ){
          mBCD = (mBCD & 0xf0) + 0x10;
     }
    return mBCD;
}

//-------------------------------------------------------------
//BCD 加 1
//入口：BCD 数据,限制的最大数，限制的最小数
//出口：BCD-1 后数据。如果==限制的最大数则表示发生溢出。
//-------------------------------------------------------------
unsigned char Math_BcdDec( unsigned char mBCD, unsigned char mLimitMax, unsigned char mLimitMin )
{
     if( mBCD == mLimitMin ){
          return mLimitMax;
```

```
    }

     mBCD--;
     if( (mBCD&0x0f) == 0x0f ){
          mBCD = mBCD - 0x0f + 0x09;
     }
    return mBCD;
}
```

13） 时间控制任务模块

显示小时和分钟，秒标志每 0.5 秒闪烁一次。在收到时间设置任务发送过来的禁止显示消息后，时间结果只在内部时间寄存器中保存，不输出到数码管显示上。在收到时间设置任务发送过来的允许显示消息后，刷新时间，并开始正常的时间显示。

ClockControl.h 文件：

```
//-----------------------------------------------------------
//实时时钟控制
//-----------------------------------------------------------
#define SECONDCOUNTMAX     50

enum TimeLocationEnum{                                  //存放时间的数组中数据存放顺序
    eSecond,
    eMinute,
    eHour
};

struct ClockControl{
    unsigned char Time[3];                              //按 BCD 码存存放的时、分、秒
};

//-----------------------------------------------------------
#ifdef _CLOCKCONTROLH
    idata struct ClockControl sClockControl;            //外部可使用的时间变量

#else
    extern idata struct ClockControl sClockControl;     //外部可使用的时间变量

    //-----------------------------------------------------------
    //初始化
    //-----------------------------------------------------------
    extern void ClockControl_Init(void);

    //-----------------------------------------------------------
    //析构
    //-----------------------------------------------------------
    extern void ClockControl_Destory(void);

    //-----------------------------------------------------------
    //主循环
    //系统时间进入一次，产生 1S 并显示
    //-----------------------------------------------------------
    extern void ClockControl_Loop(void);

    #define ClockControl_MainLoop()  do{ if(fSysTimer_Touch)ClockControl_Loop();}while(0)

#endif
```

ClockControl.c 文件：

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

```
//实时时钟控制
//-------------------------------------------------------------
#define _CLOCKCONTROLH

#include <reg52.h>
#include "Memory.h"
#include "Message.h"

#include "Math.h"
#include "DigitTube.h"
#include "DigitDrv.h"

#include "ClockControl.h"

enum TimeDisplayMode{
    eDisplaySecondFlag,                                 //秒标志，0.5 秒闪烁一次
    eDisplayMinute,                                     //显示时钟的分
    eDisplayHour,                                       //显示时钟的小时
};

//-------------------------------------------------------------
struct InClockControl{
    unsigned char mSecondCount;                         //对 20MS 计数产生 1S 的计数器
};
struct InClockControl sInClockControl;
bit fClock_DispDisable;                                 //时钟输出显示开关.=1 禁止

//-------------------------------------------------------------
//内部函数：时钟输出显示
//秒标志在 COM 的第 4、第 5 位。（D5、D6）
//分钟在 COM 的第 2、第 3 位
//小时在 COM 的第 0、第 1 位
//入口：方式（enum TimeDisplayMode）、数据
//-------------------------------------------------------------
void InClockControl_Display( unsigned char mMode, unsigned char mData )
{
    unsigned char i;

    if( fClock_DispDisable )return;       //显示输出被关闭。

    switch( mMode ){
        case eDisplaySecondFlag:          //秒标志，0.5 秒闪烁一次
            i = DigitDrv_Read(4) & ~(0x10|0x20);
            if( mData ){
                i |= 0x10|0x20;
            }
            DigitDrv_Write( 4, i );       //输出
            break;

        case eDisplayMinute:              //显示时钟的分
```

```
            DigitDrv_WriteTable( 2, mData>>4 );
            DigitDrv_WriteTable( 3, mData&0xf );
            break;

        case eDisplayHour:              //显示时钟的小时
            DigitDrv_WriteTable( 0, mData>>4 );
            DigitDrv_WriteTable( 1, mData&0xf );
            break;
    }
}

//-------------------------------------------------------------
//消息处理函数
//-------------------------------------------------------------
void InClockControl_MsgProc(void)
{
    Msg_PopMsg();                           //只此一个消息，必定 Msg_Clock_DisplayDisable
    if( Msg_PopByte() ){
        fClock_DispDisable = 1;             //显示输出关闭

        DigitDrv_Write( 4,  DigitDrv_Read(4)&~(0x10|0x20) );    //秒标志灭
    }
    else {
        fClock_DispDisable = 0;             //显示打开，要刷新

        InClockControl_Display( eDisplayMinute, sClockControl.Time[eMinute] );     //分显示
        InClockControl_Display( eDisplayHour, sClockControl.Time[eHour] );         //小时显示
    }
}

//-------------------------------------------------------------
//初始化
//-------------------------------------------------------------
void ClockControl_Init(void)
{
    Memory_Memset( (unsigned char *)&sClockControl, 0, sizeof(struct ClockControl) );
    Memory_Memset( (unsigned char *)&sInClockControl, 0, sizeof(struct InClockControl) );

    //处理的函数及消息
    Msg_Register( InClockControl_MsgProc, Msg_Clock_DisplayDisable, Msg_Clock_DisplayDisable );

    sClockControl.Time[eHour] = 0x12;               //从 12:00:00 开始
    Msg_PushByte( Msg_Clock_DisplayDisable, 0 );   //给自己发送一个消息，允许显示并刷新。
}

//-------------------------------------------------------------
//析构
//-------------------------------------------------------------
void ClockControl_Destory(void){}
```

```
//-------------------------------------------------------------
//主循环
//系统时间进入一次，产生 1S 并显示
//-------------------------------------------------------------
void ClockControl_Loop(void)
{
    if( ++sInClockControl.mSecondCount >= SECONDCOUNTMAX ){
        sInClockControl.mSecondCount = 0;
        InClockControl_Display( eDisplaySecondFlag, 1 );                    //秒标志亮.1S 到
        sClockControl.Time[eSecond] = Math_BcdInc( sClockControl.Time[eSecond], 0x59, 0x00 );

        if( sClockControl.Time[eSecond] == 0x00 )                           //1 分时间到
        {
            sClockControl.Time[eMinute] = Math_BcdInc( sClockControl.Time[eMinute], 0x59, 0x00 );
            InClockControl_Display( eDisplayMinute, sClockControl.Time[eMinute] );     //分显示

            if( sClockControl.Time[eMinute] == 0x00 )                       //1 小时时间到
            {
                sClockControl.Time[eHour] = Math_BcdInc( sClockControl.Time[eHour], 0x23, 0x00 );
                InClockControl_Display( eDisplayHour, sClockControl.Time[eHour] );    //小时显示
            }
        }
    }
    else {
        if( sInClockControl.mSecondCount == SECONDCOUNTMAX/2 ){
            InClockControl_Display( eDisplaySecondFlag, 0 );                //秒标志灭,0.5s 到
        }
    }
}
```

程序设计说明：

1）在初始化函数 ClockControl_Init 中，给自己本身这个任务模块发送了一个自我时间刷新消息：Msg_PushByte(Msg_Clock_DisplayDisable, 0)，通过一条简单的语句就实现了刷新过程，表现出的效果就是代码最大限度的复用。这样的刷新方式还有很多其它优点，比如在大程序结构中可避免外部其它硬件未初始化完成就产生输出而存在着设计隐患之类的逻辑问题，这里对此不多描述，以后会讲到。

2）本模块的消息处理函数只处理一条消息：Msg_Clock_DisplayDisable，参数为 0 时允许显示输出，并刷新全部显示。参数为 1 则关闭本模块的输出。

14） 时间设置任务模块

在编写此模块前，继续说明采用消息处理的好处。从前面可以看到。SimulantKey 按键模拟程序已经产生了时间设置模块需要的消息：Msg_Key_Setup、Msg_Key_Up、Msg_Key_Down、Msg_Key_Return，但这个模块还没有编写，整个程序都能在没有这个模块下正常运行，其它任务模块对本模块的信息只不起作用而已。当这模块加入后，这些信息才起作用。

这说明一个很重要的模块化特征：任务和任务之间的耦合性是没有的，它们之间的关联是通过系统架构支撑着。

时间设置模块和时间控制模块之间有非常弱的耦合性：只和 sClockControl. Time[3]耦合，毕竟它们合起来才真正是一个完整模块。（要实现完整的耦合性无关也是可以的，通过消息传递时间值也能实现，只是不是非常必要，毕竟合起来才叫完整时间模块）。

ClockSetup.h 文件：

```
//-----------------------------------------------------------
//实时时钟设置
//-----------------------------------------------------------

//-----------------------------------------------------------
//初始化
//-----------------------------------------------------------
extern void ClockSetup_Init(void);

//-----------------------------------------------------------
//析构
//-----------------------------------------------------------
extern void ClockSetup_Destory(void);

//-----------------------------------------------------------
//循环
//控制设置状态下的闪烁显示
//系统闪烁标志 fSysTimer_FlashTouch 成立进入
//-----------------------------------------------------------
extern void ClockSetup_Loop(void);

#define ClockSetup_MainLoop()    do{ if(fSysTimer_FlashTouch)ClockSetup_Loop();}while(0)
```

ClockSetup.c 文件：

```
//-----------------------------------------------------------
//实时时钟设置
//-----------------------------------------------------------
#include <reg52.h>
#include "SysTimer.h"
#include "Memory.h"
#include "Message.h"

#include "Math.h"
#include "DigitTube.h"
#include "DigitDrv.h"

#include "ClockControl.h"
```

```
bit fClockSetupOn;                              //=1 在设置状态

struct ClockSetup{
    unsigned char TimeBackup[3];                //按 BCD 码存存放的时、分、秒
    unsigned char mSetupId;                     //设置的位置。
};
struct ClockSetup sClockSetup;

//-------------------------------------------------------
//内部函数：时钟输出显示
//分钟在 COM 的第 2、第 3 位
//小时在 COM 的第 0、第 1 位
//入口：方式：==0 小时位置
//            ：==1 分钟
//       数据：0xff 消隐。其它时间值
//-------------------------------------------------------
void InClockSetup_Display( unsigned char mLocation, unsigned char mData )
{
    switch( mLocation ){
        case 0:  //小时位置
            if( mData == 0xff ){
                DigitDrv_Write( 0, 0 );
                DigitDrv_Write( 1, 0 );
            }
            else {
                DigitDrv_WriteTable( 0, mData>>4 );
                DigitDrv_WriteTable( 1, mData&0xf );
            }
            break;

        case 1:  //分钟位置
            if( mData == 0xff ){
                DigitDrv_Write( 2, 0 );
                DigitDrv_Write( 3, 0 );
            }
            else {
                DigitDrv_WriteTable( 2, mData>>4 );
                DigitDrv_WriteTable( 3, mData&0xf );
            }
            break;
    }
}

//-------------------------------------------------------
//消息处理
//-------------------------------------------------------
void InClockSetup_MsgProc(void)
{
    switch( Msg_PopMsg() ){
```

```
case Msg_Key_Setup:                          //设置开关键，用于开始进行时间设置和退出
    if( !fClockSetupOn ){
        //进入设置状态
        fClockSetupOn = 1;
        Msg_InsertByte( Msg_Clock_DisplayDisable, 1 ); //向ClockControl发出关闭显示。优先执行。

        //备份时间
        sClockSetup.TimeBackup[0] = sClockControl.Time[0];
        sClockSetup.TimeBackup[1] = sClockControl.Time[1];
        sClockSetup.TimeBackup[2] = sClockControl.Time[2];

        //从小时设置开始
        sClockSetup.mSetupId = 0;
    }
    else {
        //取消设置，并退回到工作界面
        fClockSetupOn = 0;
        Msg_InsertByte( Msg_Clock_DisplayDisable, 0 ); //向ClockControl发出工作显示。优先执行。
    }
    break;

case Msg_Key_Up:                        //向上减小键
    if( !fClockSetupOn )break;          //非设置状态，退出

    switch( sClockSetup.mSetupId ){
        case 0:  //设置小时
    sClockSetup.TimeBackup[eHour] = Math_BcdDec( sClockSetup.TimeBackup[eHour], 0x23, 0 );
            break;

        case 1:  //设置分钟
    sClockSetup.TimeBackup[eMinute] = Math_BcdDec( sClockSetup.TimeBackup[eMinute], 0x59, 0 );
            break;

        case 2:  //设置秒
    sClockSetup.TimeBackup[eSecond] = Math_BcdDec( sClockSetup.TimeBackup[eSecond], 0x59, 0 );
            break;
    }
    break;

case Msg_Key_Down:                      //向下增大键
    if( !fClockSetupOn )break;          //非设置状态，退出

    switch( sClockSetup.mSetupId ){
        case 0:  //设置小时
     sClockSetup.TimeBackup[eHour] = Math_BcdInc( sClockSetup.TimeBackup[eHour], 0x23, 0 );
            break;

        case 1:  //设置分钟
    sClockSetup.TimeBackup[eMinute] = Math_BcdInc( sClockSetup.TimeBackup[eMinute], 0x59, 0 );
            break;
```

```
                case 2:  //设置秒
            sClockSetup.TimeBackup[eSecond] = Math_BcdInc( sClockSetup.TimeBackup[eSecond], 0x59, 0 );
                    break;
            }
            break;

        case Msg_Key_Return:                    //确认键
            if( !fClockSetupOn )break;          //非设置状态，退出

            switch( sClockSetup.mSetupId )
            {
                case 0:  //处于小时设置->分设置
            InClockSetup_Display( 0, sClockSetup.TimeBackup[eHour] );    //保证原来设置的位置数据显示
                    sClockSetup.mSetupId++;
                    break;

                case 1:  //处于分钟设置->秒设置
                    InClockSetup_Display( 0, 0xff );                              //小时位置清除
                    InClockSetup_Display( 1, sClockSetup.TimeBackup[eSecond] );  //分钟位置显示秒
                    sClockSetup.mSetupId++;
                    break;

                case 2:  //处于秒设置->设置完毕
                    fClockSetupOn = 0;

                    //存储时间
                    sClockControl.Time[0] = sClockSetup.TimeBackup[0];
                    sClockControl.Time[1] = sClockSetup.TimeBackup[1];
                    sClockControl.Time[2] = sClockSetup.TimeBackup[2];

                    Msg_PushByte( Msg_Clock_DisplayDisable, 0 );//向 ClockControl 发出可显示。优先执行。
                    break;
            }
            break;
    }
}

//-----------------------------------------------------------
//初始化
//-----------------------------------------------------------
void ClockSetup_Init(void)
{
    fClockSetupOn = 0;
    Memory_Memset( (unsigned char *)&sClockSetup, 0, sizeof(struct ClockSetup) );
    Msg_Register( InClockSetup_MsgProc, Msg_Key_Setup, Msg_Key_Return );  //处理的函数及消息
}

//-----------------------------------------------------------
//析构
```

```
//-------------------------------------------------------------
void ClockSetup_Destory(void){}



//-------------------------------------------------------------
//循环
//控制设置状态下的闪烁显示
//系统闪烁标志 fSysTimer_FlashTouch 成立进入
//-------------------------------------------------------------
void ClockSetup_Loop(void)
{
    if( fClockSetupOn ){
        //设置状态
        switch( sClockSetup.mSetupId ){
            case 0:  //设置小时
                if( fSysTimer_FlashState ){
                    InClockSetup_Display( 0, sClockSetup.TimeBackup[eHour] );    //显示
                }
                else {
                    InClockSetup_Display( 0, 0xff );                             //消隐
                }
                break;

            case 1:  //设置分钟
                if( fSysTimer_FlashState ){
                    InClockSetup_Display( 1, sClockSetup.TimeBackup[eMinute] );
                }
                else {
                    InClockSetup_Display( 1, 0xff );
                }
                break;

            case 2:  //设置秒
                if( fSysTimer_FlashState ){
                    InClockSetup_Display( 1, sClockSetup.TimeBackup[eSecond] );
                }
                else {
                    InClockSetup_Display( 1, 0xff );
                }
                break;
        }
    }
}
```

程序设计说明：

1） fClockSetupOn 表示是否在设置状态，=1 为有效：设置中。

2） 主循环中根据设置的位置进行闪烁控制。每次系统闪烁时间标志成立则进入，闪烁状态由系统的 fSysTimer_FlashState 表示。

最后时间控制和时间设置模块的调试方法这里不列出，大家可以自己按照前面的测试用例的方式进行。

这个培训到这里基本就结束了，留有一个剩余的课题，希望大家自己在这个培训基础上完成：

要求：

1） 通过串口外部进行时间的设置。

串口通信协议格式：STX CMD Length DataBlock ChkSum ETX

其中： STX = 0x2，表示包头。

ETX = 0x3，表示包尾。

Length 表示 DataBlock 的长度

CMD 表示包的类型或格式或命令。

DataBlock 表示包的数据。

ChkSum 表示 CMD+ Length+ DataBlock 的校验和，可以用累加字节表示。

2） 每到一分钟通过串口向上发送全部时钟数据，包格式同上。

3） 考虑外部干扰，如何最有效最可靠把包解析出来。