Small RTOS(51) 1.12.1v 使用手册

概述：

Small RTOS 是一个源代码公开的多任务实时操作系统，Small RTOS 51 是其在 8051 系列处理器上的移植（使用 keil c51）。Small RTOS 可以简化那些复杂而且时间要求严格的工程的软件设计工作.本手册主要讲述 Small RTOS 的使用，本说明了在 8051 系列处理器上使用应该注意的问题。同时讲述了 Small RTOS 的移植。

本手册仅针对 Small RTOS(51) 1.12 版本

Small RTOS 的特点：

（1）公开源代码

只要遵循许可协议，任何人可以免费获得源代码。

（2）可移植

作者尽量使用把与 CPU 相关部分压缩到最小，与 CPU 无关部分用 ansi c 编写。

（3）可固化

Small RTOS 为嵌入系统设计，如果您有固化手段，它可以嵌入到您的产品中成为产品的一部分。

（4）占先式

Small RTOS 可以管理 17 个任务，用户最多可以使用 16 个任务，每个任务优先级不同。

（5）中断管理

中断可以使正在执行的任务挂起。如果优先级更高的任务被中断唤醒，则高优先级的任务在中断嵌套全部退出后立即执行。中断嵌套层数可达 255 层。如果需要，可以禁止中断嵌套管理。

（6）RAM 需求小

Small RTOS 为小 RAM 系统设计，应而 RAM 需求小，相应的，系统服务也少。

Small RTOS 的使用：

（1）系统配置：

os_cfg.h 中定义了一些常量可以配置系统，具体定义如下：

```
#define OS_MAX_TASKS          4         /* 最大任务数 1~16，也就是实际任务数                    */
#define OS_TICKS_PER_SEC      10        /* 声明 1 秒系统系统节拍数                              */
#define EN_USER_TICK_TIMER    0         /* 禁止(0)或允许(1)系统定时中断调用用户函数 UserTickTimer()  */
#define EN_OS_INT_ENTER       1         /* 禁止(0)或允许(1)中断嵌套管理                         */
#define EN_TIMER_SHARING      1         /* 禁止(0)或允许(1)定时器中断调用 OSTimeTick()          */
```

```
#define TICK_TIMER_SHARING      1          /* 定义进入多少次硬件中断为一次系统定时器软中断          */

#define EN_OS_Q                 0          /* 禁止(0)或允许(1)消息队列                              */
#define EN_OS_Q_CHK             0          /* 禁止(0)或允许(1)校验消息队列指针                      */
#define OS_Q_MEM_SEL            xdata      /* 消息队列存储空间选择，keil c51 有用，必须为 idata、xdata   */
                                /* 不是 keil c51 时它必须为空格                                  */
#define EN_OS_Q_PENT            1          /* 禁止(0)或允许(1)等待队列中的消息                      */
#define EN_OS_Q_ACCEPT          0          /* 禁止(0)或允许(1)无等待的从队列中取得消息              */
#define EN_OS_Q_POST            0          /* 禁止(0)或允许(1)FIFO 方式相对列发送消息               */
#define EN_OS_Q_POST_FRONT      1          /* 禁止(0)或允许(1)LIFO 方式相对列发送消息               */
#define EN_OS_Q_INT_POST        0          /* 禁止(0)或允许(1)中断中 FIFO 方式相对列发送消息        */
#define EN_OS_Q_INT_POST_FRONT  1          /* 禁止(0)或允许(1)中断中 LIFO 方式相对列发送消息        */
#define EN_OS_Q_NMsgs           1          /* 禁止(0)或允许(1)取得队列中消息数                      */
#define EN_OS_Q_SIZE            0          /* 禁止(0)或允许(1)取得队列总容量                        */
#define EN_OS_Q_FLUSH           0          /* 禁止(0)或允许(1)清空队列                              */

#define EN_OS_SEM               0          /* 禁止(0)或允许(1)信号量                                */
#define EN_OS_SEM_CHK           0          /* 禁止(0)或允许(1)校验信号量索引                        */
#define OS_SEM_MEM_SEL          idata      /* 信号量储空间选择，keil c51 有用，必须为 idata、xdata      */
                                /* 不是 keil c51 时它必须为空格                                  */
#define OS_MAX_SEMS             2          /* 最大信号量数目                                        */
#define EN_OS_SEM_PENT          1          /* 禁止(0)或允许(1)等待信号量                            */
#define EN_OS_SEM_ACCEPT        0          /* 禁止(0)或允许(1)无等待请求信号量                      */
#define EN_OS_SEM_INT_POST      0          /* 禁止(0)或允许(1)中断中发送信号量                      */
#define EN_OS_SEM_POST          1          /* 禁止(0)或允许(1)中发送信号量                          */
#define EN_OS_SEM_QUERY         0          /* 禁止(0)或允许(1)查询信号量                            */

//#define data                  /* 非 keil c51 时必须加此这一句                                  */


#if EN_TIMER_SHARING == 0
#define TIME_ISR_TASK_ID    SHOW_TASK_ID   /* 定义系统定时器软中断触发的任务 ID                     */
#endif
```

Os_cpu.h 中定义了一些常量可以配置一些与 cpu 相关的部分。
针对 Small RTOS 51,具体定义如下：

```
#define  OS_ENTER_CRITICAL()  EA = 0,Os_Enter_Sum++          /* 关中断                            */
#define  OS_EXIT_CRITICAL()   if (--Os_Enter_Sum==0) EA = 1  /* 开中断                            */
/* 以上两句仅可以改变“EA = ? ”部分 */

#define EN_SP2       0              /* 禁止(0)或允许(1)非屏蔽中断                                */
#define  IDATA_RAM_SIZE  0x100      /* idata 大小                                                */
#define  Sp2Space      4            /* 高级中断（软非屏蔽中断）堆栈大小 EN_SP2 为 0 时无效       */
```

```
#define OS_TIME_ISR     1                /* 系统定时器使用的中断                          */
//
SET_EA    MACRO                         ;打开所有允许中断
          SETB      EA
      ENDM
```

（2）与编译器无关的数据类型

为了便于移植，Small RTOS 定义了 6 种数据类型，它在 Os_cpu.h 定义。

针对 Small RTOS 51,具体定义如下：

```
typedef unsigned char   uint8;                   /* 定义可移植的无符号 8 位整数关键字          */
typedef signed    char  int8;                    /* 定义可移植的有符号 8 位整数关键字          */
typedef unsigned int    uint16;                  /* 定义可移植的无符号 16 位整数关键字         */
typedef signed    int   int16;                   /* 定义可移植的有符号 16 位整数关键字         */
typedef unsigned long   uint32;                  /* 定义可移植的无符号 32 位整数关键字         */
typedef signed    long  int32;                   /* 定义可移植的有符号 32 位整数关键字         */
```

（3）运行 Small RTOS

在 main 函数中调用 OSStart()后系统开始运行，OSStart()结束后运行优先级为 0 的任务。

注意：在调用 OSStart()前不要使能总中断标志，系统会自动执行开中断宏 OS_EXIT_CRITICAL()。

（4）建立任务：

Small RTOS 使用静态方法建立任务，在 config.h 中有这样一段代码：

```
#ifdef IN_OS_CPU_C
extern void TaskA(void);
extern void TaskB(void);
extern void TaskC(void);
void  (* const TaskFuction[OS_MAX_TASKS])(void)={TaskA,TaskB,TaskC};
//函数数组 TaskFuction[]保存了各个任务初始 PC 指针,其按任务 ID(既优先级次序)顺序保存
#endif
```

用户只要定义好 OS_MAX_TASKS,TaskFuction[],任务将自动运行。其中 TaskA 等为任务对应函数。

注意：1、ID 为 0 的任务为最高优先级任务,系统运行后首先执行它。

2、ID 为 OS_MAX_TASKS 的任务由系统定义。

（5）删除任务

Small RTOS 1.12 版不允许删除任务，因此，每个任务必须为无限循环结构。

（6）任务睡眠

任务可以调用 OSWait()使自己睡眠，其原型如下：

```
unsigned char OSWait(uint8 typ, uint8 ticks);
```

功能描述：系统等待函数,任务调用此函数可以等待一定时间或信号
输入：typ：等待事件类型,目前可以取以下值,或是其中任意个值的按位或
K_SIG：等待信号
K_TMO：等待超时
ticks：等待超时时的系统节拍数
输出：NOT_OK：参数错误
TMO_EVENT：超时到
SIG_EVENT：有信号
如果任务等待信号,另一个任务可以调用 OSSendSignal()使其进入就绪状态，其原型如下：

```
void OSSendSignal(uint8 TaskId);
```

功能描述：中断中给指定任务发送信号,既使指定任务就绪
输　入：TaskId：任务 ID
输　出：无
中断调用 OSIntSendSignal()也可以唤醒睡眠任务，其原型如下：

```
void OSIntSendSignal(uint8 TaskId);
```

功能描述：任务中给指定任务发送信号,既使指定任务就绪
输　入：TaskId：任务 ID
输　出：无

用户调用 OSQPend 和 OSSemPend 时也可能使任务睡眠，可参见相应章节。

（7）信号量

在 Small RTOS 中，用一个 0 到（OS_MAX_SEMS-1）的值做索引标识一个信号量，所有对信号量的访问都通过它访问。Small RTOS 在使用一个信号量之前，首先要初始化该信号量,也即调用 OSSemCreate()函数，对信号量的初始计数值赋值，该初始值为 0 到 255 之间的一个数。如果信号量是用来表示一个或者多个事件的发生，那么该信号量的初始值应设为 0。如果信号量是用于对共享资源的访问，那么该信号的初始值应设为 1（例如，把它当作二值信号量使用）。最后，如果该信号量是用来表示允许任务访问 n 个相同的资源，那么该初始值显示应该是 n，并把该信号量作为一个可计数的信号量使用。

信号量使用的内存空间由用系统分配。

（8）消息队列

与一般的 RTOS 不同，Small RTOS 的消息队列是以字节变量（uint8 型变量，范围为 0 到 255）作为消息，而不是以指针指向的内容作为消息。也就是说，消息队列发送一个消息实质是将一个 0 到 255 的数值存到消息队列中，而不是将一个指针存到消息队列中。类似的，从消息队列中获得一个消息就是得到一个范围为 0 到 255 的值。这个 0 到 255 的值用户可以任意解释。如果用户确实需要多个字节作为一个整体传递，可以有两个方法：一是消息

队列仅仅传递数据的索引，真实数据放在另外的地方；一是一次向消息队列中发送多个消息。

消息队列使用的内存空间由用户分配。

（9）Small RTOS 51 任务特殊处

由于 keil c51 由于 keil c51 默认不支持重入函数,它的重入函数使用仿真的重入栈而不使用系统栈,而 Small RTOS 51 没有进行重入栈管理，因此，用户应该保证各个任务的局部变量不会互相覆盖(方法后面有介绍)，并且不定义重入函数。

（10）中断程序编制

keil c51 例子如下：

```
#if EN_OS_INT_ENTER >0
#pragma disable                          /* 除非最高优先级中断或是不需要系统管理的中断，否则，必须加上这一句 */
#endif
   void OSTickISR(void) interrupt USER_ISR

{
   #if EN_OS_INT_ENTER >0
      OS_INT_ENTER();                    /* 中断开始处理                                    */
   #endif

   /*用户代码在这*/

   OSIntExit();                          /* 中断结束处理,                                   */
}
```

在其它 CPU 中，#pragma disable 不是必须的,但是可能需要自己在 OS_INT_ENTER()之前保存任务环境。如果某个中断不需要 OS 管理，则可以自由编写。

Small RTOS 的系统调用：

（1）OSVersion()

宏,返回 Small RTOS 版本号*100

（2）OSRunningTaskID()

宏,返回当前正在运行的任务 ID

（3）OSWait()

原型:uint8 OSWait(uint8 typ, uint8 ticks);

功能描述：系统等待函数,任务调用此函数可以等待一定时间或信号

输　入：typ：等待事件类型,目前可以取以下值,或是其中任意个值的按位或

K_SIG: 等待信号
K_TMO: 等待超时
ticks： 等待超时时的系统嘀嗒数
输 出 ：NOT_OK： 参数错误
TMO_EVENT： 超时到
SIG_EVENT： 有信号
全局变量: OSWaitTick
调用模块: OSIntSendSignal

（4）OSSendSignal()
原型:void OSSendSignal(uint8 TaskId)
功能描述: 任务中给指定任务发送信号,既使指定任务就绪
输 入: TaskId： 任务 ID
输 出: 无
全局变量: OSTaskRuning
调用模块: OSSched

（5）OSIntSendSignal()
原型:void OSIntSendSignal (uint8 TaskId);
功能描述: 中断中给指定任务发送信号,既使指定任务就绪
输 入: TaskId： 任务 ID
输 出: 无
全局变量: OSTaskRuning
调用模块: 无

（6）OSQCreate()
原型:uint8 OSQCreate(uint8 OS_Q_MEM_SEL *Buf, uint8 SizeOfBuf);
功能描述: 初始化消息队列
输 入: Buf:为队列分配的存储空间地址
SizeOfBuf:为队列分配的存储空间大小
输 出: NOT_OK:参数错误
OS_Q_OK:成功
全局变量: 无
调用模块: 无

（7）OSQPend()
原型:uint8 OSQPend(uint8 idata *Ret, uint8 OS_Q_MEM_SEL *Buf, uint8 Tick);
功能描述: 等待消息队列中的消息
输 入: Ret:返回的消息
Buf:指向队列的指针
Tick:等待时间
输 出: NOT_OK:参数错误
OS_Q_OK:收到消息
OS_Q_TMO:超时到
OS_Q_NOT_OK:无消息
全局变量: 无
调用模块: OSRunningTaskID,OSClearSignal,OSSched,OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（8）OSQAccept()

原型:uint8 uint8 OSQAccept(uint8 idata *Ret, uint8 OS_Q_MEM_SEL *Buf);

功能描述: 无等待从消息队列中取得消息

输　入: Ret:返回的消息

Buf:指向队列的指针

输　出: NOT_OK:参数错误

OS_Q_OK:收到消息

OS_Q_TMO:超时到

OS_Q_NOT_OK:无消息

全局变量: 无

调用模块: OSClearSignal,OSSched,OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（8）OSQIntPost()

原型:uint8 OSQIntPost(uint8 OS_Q_MEM_SEL *Buf, uint8 Data);

功能描述: 中断中 FIFO 方式发送消息

输　入: Buf:指向队列的指针

Data:消息数据

输　出: OS_Q_FULL:队列满

OS_Q_OK:发送成功

全局变量: 无

调用模块: OSIntSendSignal,OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（9）OSQIntPostFront()

原型:uint8 OSQIntPostFront(uint8 OS_Q_MEM_SEL *Buf, uint8 Data);

功能描述: 中断中 LIFO 方式发送消息

输　入: Buf:指向队列的指针

Data:消息数据

输　出: OS_Q_FULL:队列满

OS_Q_OK:发送成功

全局变量: 无

调用模块: OSIntSendSignal,OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（10）OSQPost()

原型:uint8 OSQPost(uint8 OS_Q_MEM_SEL *Buf, uint8 Data);

功能描述: FIFO 方式发送消息

输　入: Buf:指向队列的指针

Data:消息数据

输　出: OS_Q_FULL:队列满

OS_Q_OK:发送成功

全局变量: 无

调用模块: OSQIntPost,OSSched

（11）OSQPostFront()

原型:uint8 OSQPostFront(uint8 OS_Q_MEM_SEL *Buf, uint8 Data);

功能描述: LIFO 方式发送消息

输　入: Buf:指向队列的指针

Data:消息数据

输　出: OS_Q_FULL:队列满

OS_Q_OK:发送成功

全局变量：无

调用模块：OSQIntPostFront,OSSched

（12）OSQNMsgs()

原型:uint8 OSQNMsgs(uint8 OS_Q_MEM_SEL *Buf);

功能描述：取得消息队列中消息数

输　入：Buf:指向队列的指针

输　出：消息数

全局变量：无

调用模块：OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（13）OSQSize()

原型:uint8 OSQSize(uint8 OS_Q_MEM_SEL *Buf);

功能描述：取得消息队列总容量

输　入：Buf:指向队列的指针

输　出：消息队列总容量

全局变量：无

调用模块：OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（14）OSQFlush()

原型:void OSQFlush (uint8 OS_Q_MEM_SEL *Buf);

功能描述：清空队列

输　入：Buf:指向队列的指针

输　出：无

全局变量：无

调用模块：OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（15）OSSemCreate()

原型:uint8 OSSemCreate(uint8 index,uint8 Data);

功能描述：初始化消息队列

输　入：index:信号量索引

data:信号量初始值

输　出：NOT_OK:没有这个信号量

OS_SEM_OK:成功

全局变量：无

调用模块：无

（16）OSSemPend()

原型:uint8 OSSemPend(uint8 index, uint8 Tick);

功能描述：等待一个信号量

输　入：index:信号量索引

Tick:等待时间

输　出：NOT_OK:参数错误

OS_SEM_OK:得到信号量

OS_SEM_TMO:超时到

OS_SEM_NOT_OK:没有得到信号量

全局变量：无

调用模块：OSRunningTaskID,OSClearSignal,OSSched,OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（17）OSSemAccept()

原型:uint8 OSSemAccept(uint8 index);

功能描述: 无等待请求信号量

输　入: index:信号量索引

输　出: NOT_OK:参数错误

OS_SEM_OK:得到信号量

OS_SEM_TMO:超时到

OS_SEM_NOT_OK:没有得到信号量

全局变量: 无

调用模块: OSClearSignal,OSSched,OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（18）OSSemIntPost()

原型:uint8 OSSemIntPost(uint8 index);

功能描述: 中断中发送一个信号量

输　入: index:信号量索引

输　出: NOT_OK:参数错误

OS_SEM_OK:发送成功

全局变量: 无

调用模块: OSIntSendSignal,OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

（19）OSSemPost()

原型:uint8 OSSemPost(uint8 index);

功能描述: 发送一个信号量

输　入: index:信号量索引

输　出: NOT_OK:参数错误

OS_SEM_OK:发送成功

全局变量: 无

调用模块: OSSemIntPost,OSSched

（20）OSSemQuery()

原型:uint8 OSSemQuery(uint8 index);

功能描述: 查询信号量

输　入: index:信号量索引

输　出: 信号量的值

全局变量: 无

调用模块: OS_ENTER_CRITICAL,OS_EXIT_CRITICAL

Small RTOS 的数据结构：

（1）OSTaskRuning

定义：

```
#if OS_MAX_TASKS < 9
uint8 OSTaskRuning = 0xff;
#else
```

```
uint16 OSTaskRuning = 0xffff;
#endif
```

OSTaskRuning 存储着每一个任务状态（就绪/运行还是休眠），每一个任务存储一位，1 代表就绪/运行，0 代表休眠。OSTaskRuning 的最低位存储 ID 为 0 的任务状态，次低位存储 ID 为 0 的任务状态。以此类推。

（2）OSWaitTick

定义：

```
uint8 OSWaitTick[OS_MAX_TASKS];
```

OSWaitTick 存储各个任务剩余等待系统节拍数。

（3）OSIntNesting

定义：

```
#if EN_OS_INT_ENTER > 0
uint8 OSIntNesting;
#endif
```

OSIntNesting 存储中断嵌套层数。

（4）OSTaskID

定义：

```
uint8 OSTaskID;
```

OSTaskID 存储当前运行任务的 ID。

（5）OSNextTaskID

定义：

```
uint8 OSNextTaskID;
```

OSNextTaskID 用于标明将要运行的任务的 ID

（6）Os_Enter_Sum

定义：

```
uint8 Os_Enter_Sum=0;
```

Os_Enter_Sum 是 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()使用的信号量。

（7）消息队列数据结构

定义（用户程序中）：

```
uint8 OS_Q_MEM_SEL SerialData[n];
```

任务数小于 9 个时：
Buf[0]:队列中字节数,
Buf[1]:Buf 总长度
Buf[2]:出对端,
Buf[3]:等待队列任务列表
Buf[4]~Buf[n-1]:存储消息

任务数大于 8 个时：
Buf[0]:队列中字节数,
Buf[1]:Buf 总长度
Buf[2]:出对端,
Buf[3]、Buf[4]:等待队列任务列表
Buf[5]~Buf[n-1]:存储消息

（8）信号量数据结构
定义：

```
#if OS_MAX_TASKS < 9
uint8 OS_SEM_MEM_SEL OsSemBuf[OS_MAX_SEMS*2];
#else
uint8 OS_SEM_MEM_SEL OsSemBuf[OS_MAX_SEMS*3];
#endif
```

任务数小于 9 个时(index 为 0~(OS_MAX_SEMS-1))：
OsSemBuf[index * 2]:信号量的值
OsSemBuf[index * 2 + 1]:等待信号量的任务列表

任务数小于 9 个时(index 为 0~(OS_MAX_SEMS-1))：
OsSemBuf[index * 3]:信号量的值
OsSemBuf[index * 3 + 1]、OsSemBuf[index * 3 + 2]:等待信号量的任务列表

Small RTOS 51 的特殊的数据结构：
（1）OSFastSwap
定义：

```
#if OS_MAX_TASKS < 9
unsigned char data OSFastSwap=0xff;
#else
unsigned int data OSFastSwap=0xffff;
#endif
```

OSFastSwap 存储着每一个任务如何被切换（自己调用 OSSchedt 还是中断程序激活更高级任务），每一个任务存储一位，1 代表因为自己调用 OSSched 而被切换，0 代表因为中断程序激活更高级任务而被切换。最低位存储 ID 为 0 的任务状态，次低位存储 ID 为 0 的任务状态。以此类推。

如果任务因为自己调用 OSSched 而被切换，则需要存储的任务环境所需内部 RAM 少的多。

（2）OSTsakStackBotton

定义：

```
unsigned char idata * OSTsakStackBotton[OS_MAX_TASKS+2];/* 任务堆栈底部位置          */
```

OSTsakStackBotton 存储各个任务的栈底和栈顶位置。因为后一个任务的栈底是前一个任务的栈顶，因此其个数为 OS_MAX_TASKS+2（任务数量为 OS_MAX_TASKS+1，其中有一个隐含任务）。考虑到第一个任务的栈底和最后一个任务的栈顶的值在运行时实际不会改变，OSTsakStackBotton 可以减少两个字节空间占用，但这样程序复杂一些。

Small RTOS 的移植：

（1）在 os_cpu.h 中定义几个宏（以 keil c51 为例）：

```
#define  OS_INT_ENTER() OSIntNesting++                    /* 中断嵌套管理          */
#define  OS_ENTER_CRITICAL()  EA = 0,Os_Enter_Sum++          /* 关中断                */
#define  OS_EXIT_CRITICAL()   if (--Os_Enter_Sum==0) EA = 1    /* 开中断                */

#define  HIGH_BYTE  0                                 /* uint16 的高位字节       */
#define  LOW_BYTE   1                                 /* uint16 的低位字节       */

#define  OS_TASK_SW()       OSCtxSw()                 /* 任务切换函数          */
```

OS_ENTER_CRITICAL()、OS_EXIT_CRITICAL()分别定义为关中断和开中断在特定的 c 编译器的表示方法。

HIGH_BYTE、LOW_BYTE 定义 uint16 型变量在特定的 c 编译器的存储方法，如果高位字节的地址小于低位字节的地址（如 keil c51），则 HIGH_BYTE 为 0，LOW_BYTE 为 1。否则（如 8086 系列），HIGH_BYTE 为 1，LOW_BYTE 为 0。

OS_INT_ENTER 则将变量 OSIntNesting 加 1。它仅在中断服务程序中使用。

注意：如果中断嵌套层数可能超过 255，OS_INT_ENTER 要防止 OSIntNesting 溢出。

OS_TASK_SW()定义非中断中任务切换时执行的指令，可以是一条软中断指令（例如在 8086 系列 CPU 上），或仅仅是函数调用（如 keil c51）。

（2）定义与编译器无关的变量类型（以 keil c51 为例）：

```
typedef unsigned char   uint8;                   /* 定义可移植的无符号 8 位整数关键字       */
typedef signed    char  int8;                    /* 定义可移植的有符号 8 位整数关键字       */
typedef unsigned int    uint16;                  /* 定义可移植的无符号 16 位整数关键字      */
typedef signed    int   int16;                   /* 定义可移植的有符号 16 位整数关键字      */
```

```
typedef unsigned long  uint32;                    /* 定义可移植的无符号 32 位整数关键字        */
typedef signed   long  int32;                     /* 定义可移植的有符号 32 位整数关键字        */
```

（3）在 os_cpu_c.c 和 os_cpu_a.asm 中定义几个函数：
OSStart、OSIntCtxSw、OSTickISR、OSIdle 和 OS_TASK_SW()最终调用的函数或中断。

OSStart:初始化任务并让 ID 为 0 的任务执行。同时允许中断。
定义如下：

```
    void OSStart(void)

{
   初始化除 ID 为 0 以外所有任务堆栈;
   OSTaskID = 0;
   使堆栈指针指向 ID 为 0 的任务堆栈空间;
   OS_EXIT_CRITICAL();
   使程序指针指向 ID 为 0 的任务的程序首地址;
}
```

OSIntCtxSw:中断中任务切换函数
定义如下：

```
    void OSIntCtxSw(void)

{
   堆栈指针调整为中断程序调用 OSIntExit 前的状态;
   堆栈空间变换;
   堆栈指针指向新的堆栈;
   OSTaskID = OSNextTaskID;
   恢复任务环境;
   中断返回指令;
}
```

OSIntCtxSw 由 OSIntExit 直接调用，堆栈指针调整为中断程序调用 OSIntExit 前的状态即为执行若干出栈指令。
堆栈空间变换可以参照 keilc51 目录下 Os_cpu_c.c 文件中被注释的 C_OSCtxSw 函数，C_OSCtxSw 还包括 堆栈指针指向新的堆栈.

OSTickISR 为系统节拍中断服务程序
定义如下：

```
    void OSTickISR(void)

{
```

```
#if TICK_TIMER_SHARING >1
    static unsigned char TickSum=0;
#endif

    禁止中断;
    保存任务环境;

#if TICK_TIMER_SHARING >1
    TickSum = (TickSum + 1) % TICK_TIMER_SHARING;
    if (TickSum != 0)
    {
        允许中断;
        恢复任务环境;
        return;
    }
#endif


#if EN_OS_INT_ENTER >0
    OS_INT_ENTER();                         /* 中断开始处理                    */
#endif
    允许中断;

#if USER_TICK_TIMER_EN == 1
    UserTickTimer();                        /* 用户函数                        */
#endif

#if EN_TIMER_SHARING > 0
    OSTimeTick();                           /* 调用系统时钟处理函数            */
#else
    OSIntSendSignal(TIME_ISR_TASK_ID);
#endif

    OSIntExit();                            /* 中断结束处理                    */
}
```

其中有一些 CPU 的禁止中断和保存任务环境由 CPU 自动处理或是自动处理一部分。

OSIdle()优先级最低的任务
定义如下：

```
    void OSIdle(void)
{
```

```
    while(1)
    {
       /* 使 CPU 处于省电状态 */

    }
}
```

OS_TASK_SW()最终调用的函数或中断：非中断中任务切换函数
定义如下：

```
        void OS_TASK_SW(void)

    {
       保存任务环境;
       堆栈空间变换;
       堆栈指针指向新的堆栈;
       OSTaskID = OSNextTaskID;
       恢复任务环境;
       恢复程序指针;
    }
```

Small RTOS 51(for keil)的特殊说明：

编译器版本需求需求

当不使用消息队列时,需要 Keil C51 V6.14 以上版本。

当使用消息队列时,需要 Keil C51 V7.00 以上版本。

C 语言优化等级设置

优化等级设置不能大于 7，可以等于 7。

目标系统需求

Small RTOS 51 可以在没有任何外部数据存储器的单片 8051 系统上运行但应用程序仍然可以访问外部存储器。Small RTOS 51 可以使用 C51 支持的全部存储器模块，选择记忆模型仅影响应用目标的位置。一般来说 Small RTOS 51 应用程序工作在小模式下。Small RTOS 51 没有按照 bank switching 程序设计，不能使用 code banking 程序。

可再入功能

不允许从几个任务或中断过程调用非可再入 C 语言函数。

非可再入 C51 函数将它们的参数和自动变量局部数据保存在静态存储器内因此当重复调用函数时这些数据会被改写。非可再入 C 语言函数不可第归调用，不可被多个任务同时调用，不可被一个或多个任务与一个或多个中断同时调用。Small RTOS 51 系统函数不会调用任何这样的函数。那些仅使用寄存器作为参变量和自动变量的 C 语言函数总是可再入的而且可以从不同的 Small RTOS 51 任务中没有任何限制的调用。C51 编译程序也提

供可再入功能，参看“ C51 用户手册”以便获得更多信息。可再入函数（用 reentrant 关键字的函数）将他们的参变量和局部数据变量储存到一个可再入堆栈内并且数据是被保护的以预防多重呼叫。然而如果你在你的应用程序中使用可再入函数（用 reentrant 关键字的函数）你必须保证这些功能不呼叫任何 Small RTOS 51 系统函数。而且那些可再入函数（用 reentrant 关键字的函数）不会被 Small RTOS 51 任务调度所中断。特别注意一点，可再入堆栈不得放在内部 RAM 中。

C51 库函数

全部的可再入 C51 库函数可以没有任何限制的用于全部任务。非可再入 c51 库函数与非可再入 C 语言函数在应用时有着同样的限制。

多数据指针和数学单元的用法

c51 编译程序允许你使用 8051 派生类型的多数据指针和数学单元。因为 Small RTOS 51 不包括任何对这些硬件的管理，最好你不要与 Small RTOS 51 一起使用这些器件。如果你可以保证在使用这些派生硬件的程序执行期间不会被任务调度中断的话你可以使用多数据指针和数学单元。

寄存器段

Small RTOS 51 分配全部任务到寄存器段 0。因此全部的任务函数必须用 c51 的默认设置 registerbank 0 编译。不需要 Small RTOS 51 管理的中断函数可以使用剩余的寄存器段。

局部变量

keil c51 采用变量覆盖的方法分配局部变量，而不是把局部变量分配到堆栈中，当在 keil c51 使用 Small RTOS 时，编译系统会把各个任务的局部变量分配在同一块内存，造成程序运行错误。因此，最多只能允一个任务函数与?CO?OS_CPU_C 进行覆盖分析，禁止任务函数与?CO?OS_CPU_C 进行覆盖分析的方法如下：

在 Progect->BL51 Misc->Overlay 里填"?CO?OS_CPU_C~任务函数名,"

每一项禁止一个任务函数。

如果任务（包括任务调用的函数）没有局部变量，可以不进行以上处理。

注意参数也是局部变量。

另外，用户函数被多个任务或和中断调用且不可再入（程序保证各个任务和中断不同时调用此函数），它的局部变量不能与任何任务的局部变量覆盖，应该禁止所有调用它的任务与之进行覆盖分析，方法与上面一样：

在 Progect->BL51 Misc->Overlay 里填"任务函数名~用户函数名,"。

在进行以上处理后，会出现多条类似如下的警告：

*** WARNING L16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS
SEGMENT: ?PR?XXXX?XXX

不用理会它.

注意，以上最后一项不需要逗号。

关于软非屏蔽中断

当某个中断对时间要求非常严格的时候,用户可以使用软非屏蔽中断,使

之不受 Small RTOS 51 关中断的影响，但它也不再受 OS 管理。设置方法如下：

（1）将这个中断设置为最高优先级中断，其它受 OS 管理的中断优先级必须比它低。因此，标准 51 系列单片机就不需要中断嵌套管理了。有些 51 单片机有超过 2 个中断优先级，它们还需要中断嵌套管理。

（2）将 OS_CPU.h 中 EN_SP2 定义为 1

（3）将 OS_CPU.h 中宏 OS_ENTER_CRITICAL()定义为类似

```
IE = IE & (~0x30)
```

形式，其中 0x30 根据程序要求取值，为 1 的位为程序运行时需要打开且受 OS 管理的中断。

（4）将 OS_CPU.h 中宏 OS_ENTER_CRITICAL()定义为类似

```
IE = IE | 0xb0
```

形式，其中 0xb0 根据程序要求取值，为 1 的位为程序运行时需要打开且受 OS 管理的中断。

（5）将 OS_CPU.h 中 Sp2Space 定义合适大小（ISR 使用堆栈最大值+2）。

（6）将 OS_CPU.h 中宏 SET_EA 定义为类似

```
orl     IE,#0b0h
```

形式，其中 0b0h 根据程序要求取值，以允许程序运行时受 OS 管理的中断可以中断 CPU。

（7）在合适的时候允许或禁止软非屏蔽中断。

（8）软非屏蔽中断不允许调用 OS_INT_ENTER()和 OSIntExit()，因此，如果在软非屏蔽中断中调用 OSIntSendSignal()使高优先级任务就绪，OS 也不进行任务切换。作者不推荐在软非屏蔽中断中调用 OSIntSendSignal()。

其它注意事项

1、通过调用系统函数进行任务切换，保存任务环境需要 RAM 空间为（2+到 OSSched 时的调用层次*2）字节。通过中断进行任务切换保存任务环境需要 RAM 空间为（中断发生时堆栈使用量+15）字节，这 15 字节包括 R0~R7,ACC,B,PSW,DPH,DPL 和返回地址。而 8051 系列 idata 小（51 只有 128 字节，52 只有 256 字节），因此，当任务较多时，应该避免过多的通过中断进行任务切换。

2、OS_CPU.H 中的 IDATA_RAM_SIZE 应当与实际的 idata 相同。