[SMC-RTOS 编程指南]

摘要

[通过本篇文章让你了解 SMC-RTOS 由来及设计原理
使你快速使用和了解接口函数]

宋牧春

[scdef@163.com]

目录

第1章 简介

1.1 SMC-RTOS 由来

作为一名单片机的爱好者，我想大家都沉迷于自己作为造物主的角色。控制着各种形形色色的传感器，实现各种各样的功能。然而，在编程的世界中，你是否还在一直使用前后台轮询的方式编程。站在自己的角度来说，我一直是。当需要很多个线程的时候，我一般采取的方法是在定时器中为每一个线程定义一个计数器和一个事件标志位。当计数器达到设定的时间就立刻置位相应的标志位。main 函数中通过查询置位的标志位进行线程处理。我想，聪明的你肯定明白我在说什么。但是，每一次多以线程就要定义一个事件标志位。命名都烦，因此就萌发使用操作系统的念头。但是经过自己的思考，我觉得现有的 uC/OS-II 是资料最多的 RTOS，因此编程方便。但是，uC/OS-II（商业软件）又感觉过于臃肿并且开源但是不免费。因为我只想有一个线程调度的功能即可。所以，就决定自己写一个 RTOS，仅仅包含基本的线程调度、时间片轮转以及信号量同步等功能。当然啦，更多的功能，完全可以由你自己完成或者日后我来完成，供大家选择。

1.2 致谢

本人由于看过 Linux 内核、U-Boot、uC/OS-II 以及 RT-Thread 的等工程源代码，因此在编程的过程中，或多或少的借鉴了他们优秀的代码风格以及编程思想。本人非常不喜欢 uC/OS-II 的代码风格以及命名规范，因此这也是我不喜欢 uC/OS-II 的原因之一，但是优秀的代码的思想值得借鉴。本人英语能力有限，因此在命名上为了更加符合英语的表达，因此在函数和变量的命名上部分直接采用 RT-Thread 的命名。在链表的使用，借鉴了 Linux 内核的使用方法，Linux 内核对链表的操作可谓是我见到的代码之中最优秀之一。因此，在这里感谢这些优秀 programmer。

因此，我对于 SMC-RTOS 的定位是：SMC-RTOS 是一款开源嵌入式实时操作系统（遵循 GPL 许可协议），它短小精悍，主要包含线程调度、信号量以及软件定时器。现阶段已经移植成功的平台有 Cortex-M3 和 Cortex-M4 内核的 SoC。

1.3 源码获取

SMC-RTOS 是完全开源的实时操作系统，为了让更多感兴趣的人使用和学习，SMC-RTOS 的全部源码可以从 github 链接：https://github.com/smcdef/SMC-RTOS 下载。感谢你的使用和支持，如果使用发现任何的 bug 可以发送到我的邮箱：smcdef@163.com。

1.4 目标人群

SMC-RTOS 功能简单，代码量不多，因此适合想使用操作系统而又只想使用 RTOS 的线程调度功能。由于代码量少，因此可以更简单的阅读源码掌握调度的整个过程，在系统调试的时候，可以帮助我们确定问题存在是用户代码部分还是 RTOS 部分。同时，也适合新手学习，麻雀虽小，可是五脏俱全啊！通过阅读源码可以让你掌握 RTOS 的工作原理。

第2章 内核结构

2.1 线程调度

线程（thread）的调度代码都在 smc_thread.c 文件中实现，这里所说的线程就相当于 uC/OS-II 中的线程，两者意思完全一样。SMC-RTOS 的调度器是基于优先级的全抢占式调度。系统最多支持 32 个优先级（0~SMC_THREAD_MAX-1，0 为优先级最高，SMC_THREAD_MAX-1 分配给空闲线程）。优先级支持的数目可以在 smc_config.h 文件中定义 SMC_THREAD_MAX 的值。在系统中，当有比当前线程优先级高的就绪线程的时候，系统就会切换到最高优先级的线程执行。系统支持相同优先级的线程，同一优先级的线程会进行时间片轮转运行。时间片大小由用户设定。

SMC-RTOS 中维护了一个线程优先级就绪队列，如图 2-1 所示。在 SMC-RTOS 中定义了一个拥有 SMC_THREAD_MAX（最大 32）个元素的线程优先级数组 smc_list_head_table，数组中每一个元素都是同一优先级线程链表的头结点，相同优先级的线程形成一个双向链表。在发生线程调度的时候，如果同一优先级仅仅存在一个线程，则直接切换到当前就绪的线程。如果多个线程具有相同的优先级，则切换当前优先级链表头结点指向的下一个线程，在当前线程时间片使用完毕的时候，将当前线程的线程控制块从双向链表中移除，并添加到该优先级链表头结点的尾部，然后同样切换到当前优先级链表头结点指向的下一个线程运行。

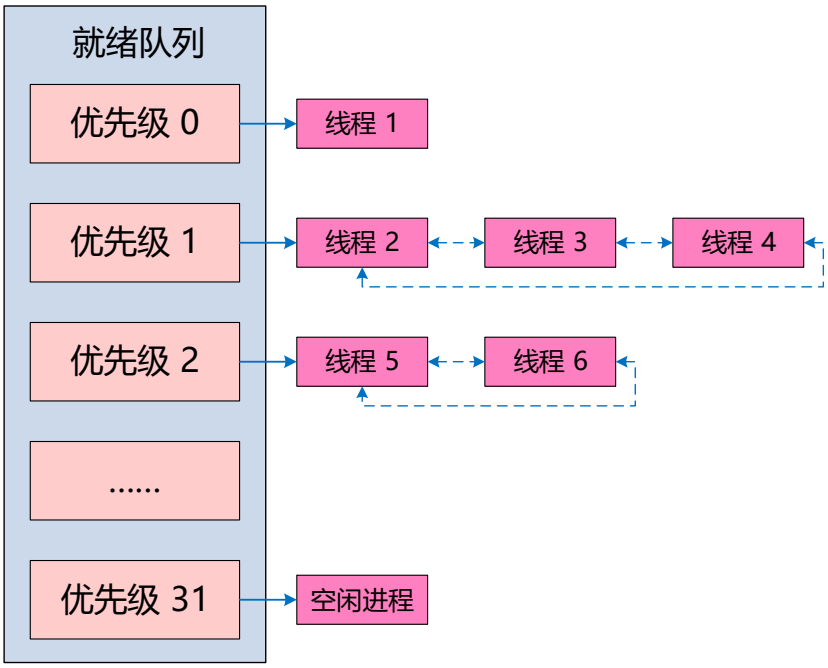

图 2-1 线程就绪示意图

在图 2-1 中，优先级 1 的线程有线程 2、线程 3 以及线程 4。线程 2 的执行必须等到线程 1 挂起，线程 3 的执行必须要等到线程 1 挂起以及线程 2 的时间片用完。最低优先级分配给空闲线程。

在 SMC-RTOS 中定义了两个全局变量 smc_thread_current 和 smc_thread_ready 分别用来存放当前运行线程的线程控制块的地址和就绪最高优先级的线程控制块地址。系统初始化完成后，执行第一次线程调度之前，smc_thread_current 的值为 0，smc_thread_ready 指向就绪队列中最高优先级线程的控制块。

2.2 空闲任务

当系统中不存在任何就绪的线程的时候，CPU 就会运行最低优先级的空闲线程。当用户使用系统 CPU 使用率计算功能的时候，需要在 smc_config.h 文件中定义 SMC_USING_CPU_USAGE。如果使用 CPU 使用率计算模块，那么系统启动后运行的第一个线程将不再是最高优先级的就绪线程，而是空闲线程。在空闲线程中，smc_cpu_usage_init( )函数将会启动一个 100ms 的定时器并调用 smc_scheduler_lock( ) 进行调度上锁，此时在空闲线程中对计数器 smc_idle_cnt_run 进行递增操作。直到定时时间到来，执行定时器 smc_idle_timer 的超时函数 smc_idle_timeout( )。在超时函数中，获取空闲线程在没有任何线程切换的情况下计数器的最大值并改变定时器时间为 1 秒，然后调度解锁。每隔 1 秒钟，计算一次 CPU 的使用率，可以通过调用如下接口函数函数返回 CPU 使用率，精度 1%。

```
smc_uint8_t smc_get_cpu_usage(void);
```

2.3 编程 API

2.3.1 线程创建

创建一个线程首先需要定义线程使用的栈以及线程控制块，可以使用 smc_stack_t 定义一个数组作为线程使用的栈。使用 smc_thread_t 定义一个线程的控制块，该控制块包含了当前线程运行所需要的所有信息。然后调用如下函数初始化线程。

```
void smc_thread_init(smc_thread_t *thread,
                     void (*entry)(void *parameter),
                     void *parameter,
                     smc_uint8_t priority,
                     void *stack_start,
                     smc_uint32_t stack_size,
                     smc_uint32_t slice_tick);
```

函数参数如表 2-1 所示。

表 2-1 函数参数

参数	描述
thread	线程控制块指针
entry	线程入口函数
parameter	线程入口函数参数
priority	线程优先级（0~SMC_THREAD_MAX-1）
stack_start	线程栈起始地址
stack_size	线程栈字节大小
slice_tick	线程时间片大小

线程创建使用举例如下：

```
#define LED_THREAD_PRIORITY    0
static smc_uint8_t led_stack[512];            /* led thread stack */
static smc_thread_t led_thread;               /* led thread structure */


static void led_thread_entry(void *parameter)
{
    /* initialize led hardware */
    smc_hw_led_init();

    while (1) {
        /* led on */
        smc_hw_led_on(0);
        /* sleep 0.5 second and switch to other thread */
        smc_thread_delay(SMC_TICKS_PER_SECOND / 2);

        /* led off */
        smc_hw_led_off(0);
        /* sleep 0.5 second and switch to other thread */
        smc_thread_delay(SMC_TICKS_PER_SECOND / 2);
    }
}

void smc_app_init(void)
{
    /* init led thread */
    smc_thread_init(&led_thread,
                    led_thread_entry,
                    NULL,
                    LED_THREAD_PRIORITY,
```

```
                    led_stack,
                    sizeof(led_stack),
                    20);
}
```

线程入口函数是一个死循环函数，该线程函数第一次执行的时候会调用 smc_hw_led_init 初始化 led 的硬件。因此，我们可以将硬件的初始化代码放在循环开始执行的前面保证只被调用一次。我么可以将所有的线程创建代码放在 smc_app_init 函数中。

2.3.2 线程延时

任何的线程在创建之后就会参与线程调度，但是如果线程不主动让出 CPU 的话，就无法使低于该线程优先级的线程运行。线程在执行的过程中如果想让出当前线程的 CPU 一段时间可以调用如下函数。

```
void smc_thread_delay(smc_uint32_t delay_tick);
```

函数参数如表 2-2 所示。

表 2-2 函数参数

参数	描述
delay_tick	线程延时时间

延时时间只能是系统滴答定时器最小分辨率的整数倍。调用该函数后会将当前线程挂起并启动一次软件定时器，然后执行线程调度，让出 CPU 使用权给就绪队列中最高优先级线程。该函数的参数指定延时时间大小，当指定时间达到后，从挂起状态切换到就绪态并插入就绪队列链表。当前线程优先级在就绪线程中最高时，该线程将会重新得到运行。

2.3.3 线程挂起

线程初始化的时候默认是就绪态并插入就绪链表中等待线程调度。当线程处于挂起状态的时候就不会参与系统线程调度，直到用户调用相关函数恢复线程。如果想挂起某一个线程可以调用如下函数。

```
smc_int32_t smc_thread_suspend(smc_thread_t *thread);
```

函数参数如表 2-3 所示。

表 2-3 函数参数

参数	描述
thread	即将被挂起的线程控制块指针

如果当前线程调用挂起自身的情况下，在执行挂起之后必须调用系统调度函数 smc_scheduler( )。如果当前线程挂起其他线程则不需要调用系统调度函数。

2.3.4 线程恢复

当一个线程处于挂起状态的时候，可以通过线程恢复接口将线程从挂起线程链表中移除并添加到就绪链表中参与线程调度。当被恢复的线程的优先级高于当前运行线程时，系统会立即执行上下文切换。线程恢复使用如下函数接口。

```
smc_int32_t smc_thread_resume(smc_thread_t *thread);
```

函数参数如表 2-4 所示。

表 2-4 函数参数

参数	描述
thread	即将被恢复的线程控制块指针

2.3.5 调度上锁

当前线程进入临界区的时候，我们除了关闭系统总中断以外还可以给调度上锁，当调度上锁之后，系统中断仍然响应。但是，系统不会进行上下文切换，必须等到调度解锁。关闭总中断适合执行时间短的情况，因为在实时操作系统中，尽量避免关中断。而调度上锁适合执行时间长。调度上锁接口如下。

```
void smc_scheduler_lock(void);
```

2.3.6 调度解锁

当调度上锁之后，必须调用调度解锁接口，否则系统无法进行线程切换。调度上解锁接口如下。

```
void smc_scheduler_unlock(void);
```

2.3.7 空闲线程

当所有高于空闲线程优先级的线程执行完毕时，系统就会切换到空闲线程。我们可以设置钩子函数，当空闲线程运行的时候执行用户钩子函数。钩子函数中可以使 CPU 休眠或者处理低功耗，尽量保证钩子函数的执行时间短。用户设置钩子函数可以使用如下接口函数。

```
void smc_thread_idle_sethook(void (*hook)(void));
```

函数参数如表 2-5 所示。

表 2-5 函数参数

参数	描述
hook	空闲线程接口函数

2.3.8 调度钩子

设置调度钩子函数，在系统线程切换时，这个钩子函数将被调用，设置接口函数如下。

```
void smc_scheduler_sethook(void (*hook)(void));
```

函数参数如表 2-6 所示。

表 2-6 函数参数

参数	描述
hook	线程调度接口函数

第3章 定时器

定时器一般分为软件定时器和硬件定时器。SMC-RTOS 利用系统滴答时钟进行软件模拟定时器。但是，定时时间只能为系统节拍的整数倍。例如节拍是 5ms，软件定时器的定时间隔只能是 5ms，10ms，15ms 等 5 的整数倍，不能是 6ms，12ms 等。SMC-RTOS 提供两种定时器工作模式，分别为单次定时和周期定时。单次定时的情况下，当软件定时时间达到时，软件定时器停止工作。周期定时器在软件定时器时间到达的时候重新开始计时，周期性的定时。

SMC-RTOS 中的软件定时器中断服务函数是在中断中执行的。因此编程者应该保证软件定时器入口函数的代码执行时间尽量短。并且，在函数内部不应该导致当前上下文挂起或者等待。

3.1 定时器管理

SMC-RTOS 中维护一个软件定时器等待链表头指针 smc_timer_list。所有的软件定时器都将会被插入双向链表中，其中包括软件定时器以及线程延时。实际上，线程在延时的时候实际上就相当于一个单次软件定时器。因此，可以把两者同等对待。

当创建一个软件定时器的时候，在启动软件定时器之后，就会将该定时器插入双向链表。插入的时候按照定时时间从小到大的顺序排序。并且第 n 个插入的定时器的延时时间是前 n-1 个定时器延时时间之和。链表中只存在一个定时器 timer1 并且定时 5 个系统节拍的示意如图 3-1 所示。

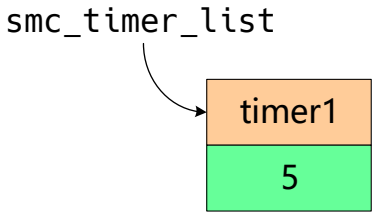

图 3-1 一个定时器链表情况

当继续创建一个定时 8 个系统节拍定时器 timer2 的时候，插入链表的情况如图 3-2 所示。

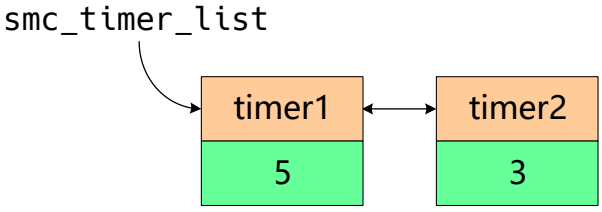

图 3-2 插入 timer2 链表示意图

插入的 timer2 定时设定的定时时间节拍是 8 个，但是插入链表之后时间上是减去 timer1 的定时节拍。现在我们继续插入定时器 timer3，假设 timer3 定时 6 个系统节拍，则插入后链表示意图如图 3-3 所示。我们可以看到 timer2 和 timer3 的定时节拍都发生了变化。timer3 的节拍等于 timer1 加上 timer3 之和，依然是 6 个节拍。timer2 的节拍等于 timer1 加上 timer3 加上 timer2，依然是 8 节

拍。之所以这么做的原因就是，在系统节拍中断中不需要对链表中的每一个定时器进行减 1 操作，只需要对链表中的第一个定时器进行减 1 操作，因此节省了节拍中断的执行时间。

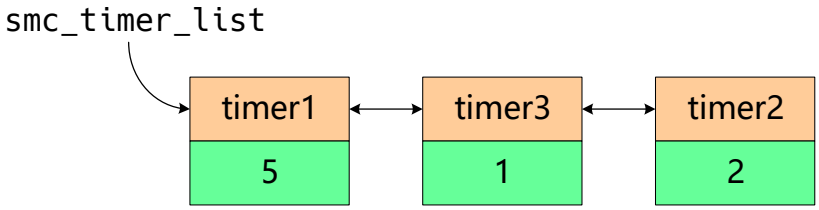

图 3-3 插入 timer3 链表示意图

3.2 编程 API

3.2.1 定时器创建

软件定时器的工作方式有两种方式 :单次定时器和周期定时器。定时器的创建可以调用如下接口函数。

```
void smc_timer_init(smc_timer_t *timer,
                    smc_uint32_t tick,
                    void (*timerout)(void *parameter),
                    void *parameter,
                    smc_uint8_t flag);
```

函数参数如表 3-1 所示。

表 3-1 函数参数

参数	描述
timer	软件定时器控制块指针
tick	定时时间大小
timeout	定时时间到达时入口函数
parameter	定时器入口函数参数
flag	定时器工作方式标志。 SMC_TIMER_ONCE：单次定时； SMC_TIMER_PERIODIC：周期定时。

3.2.2 定时器启动

当软件定时器创建后需要启动定时器才能工作，启动定时器可以调用如下接口函数。

```
void smc_timer_enable(smc_timer_t *timer);
```

函数参数如表 3-2 所示。

表 3-2 函数参数

参数	描述
timer	即将启动的定时器控制块指针

当定时器被启动的时候，系统会将定时器插入定时就绪链表中。

3.2.3 定时器关闭

软件定时器启动之后可以调用相关的接口关闭，但是必须保证关闭的定时器之前状态是打开的，关闭定时器可以调用如下接口函数。

```
void smc_timer_disable(smc_timer_t *timer_del);
```

函数参数如表 3-3 所示。

表 3-3 函数参数

参数	描述
timer_del	即将关闭的定时器控制块指针

当定时器被关闭的时候，系统会将定时器定时就绪链表中删除。

定时器程序举例如下：

```
static smc_timer_t timer1;
/* timer1 timeout entry function */
static void timeout1(void *parameter)
{
    /* do something! */
    led0 = !led0;
}

void smc_app_init(void)
{
    /* init timer1 for 500ms with periodic*/
    smc_timer_init(&timer1,
                   SMC_TICKS_PER_SECOND / 2,
                   timeout1,
                   NULL,
                   SMC_TIMER_PERIODIC);

    /* timer1 start */
    smc_timer_enable(&timer1);
}
```

当我们进行嵌入式软设计的时候，往往需要一个系统指示灯闪烁表示系统正在运行。假设，LED 闪烁频率 1Hz，那么可以使用软件定时器定时 500ms，在定时器入口函数中驱动 LED 闪烁。与使用一个线程去驱动 LED 灯闪烁相比，软件定时器可以节省系统 RAM 资源以及降低 CPU 使用资源。

第4章 线程通信

4.1 信号量

信号量（semaphore）是进程间通信处理同步互斥的机制。是在多线程环境下使用的一种措施，它负责协调各个进程，以保证他们能够正确、合理的使用公共资源。

4.1.1 控制块

```
/**
 * Semaphore structure
 */
typedef struct smc_sem {
    smc_list_head_t slist;     /* Thread that is suspended for waiting for
a semaphore */
    smc_uint16_t    value;      /* semaphore value */
} smc_sem_t;
```

slist 是所有等待该信号量而挂起线程链表的头结点，value 是信号量的值，最大 65535。我们可以可以将信号量的 value 当作资源的数量。

4.1.2 编程 API

4.1.2.1 信号量创建

使用 smc_timer_t 类型定义一个信号量，然后调用如下接口函数初始化。

```
void smc_sem_init(smc_sem_t *sem, smc_uint16_t value);
```

函数参数如表 4-1 所示。

表 4-1 函数参数

参数	描述
sem	需要初始化信号量的指针
value	信号量的值

4.1.2.2 信号量获取

线程通过获取信号量来获得信号量资源实例，当信号量值大于零时，线程将获得信号量，并且相应的信号量值都会减 1，获取信号量使用下面的函数接口。

```
smc_int32_t smc_sem_pend(smc_sem_t *sem, smc_int32_t time_out);
```

在调用这个函数时，如果信号量的值等于零，那么说明当前信号量资源实例不可用，申请该信号量的线程将根据 time_out 参数的情况选择直接返回、或挂起等待一段时间、或永久等待，直到其他线程或中断释放该信号量。

函数参数如表 4-2 所示。

表 4-2 函数参数

参数	描述
sem	获取信号量对象
time_out	等待超时时间 SMC_SEM_NO_WAIT:不等待，立即返回（不管是否获取成功） SMC_SEM_WAIT_FOREVER:永远等待，直到获取信号量对象

4.1.2.3 信号量释放

当线程完成资源的访问后，应尽快释放它持有的信号量，使得其他线程能获得该信号量。释放信号量使用下面的函数接口。

```
smc_int32_t smc_sem_release(smc_sem_t *sem);
```

当信号量的值等于零时，并且有线程等待这个信号量时，将唤醒等待在该信号量线程队列中的第一个线程，由它获取信号量。否则将把信号量的值加 1。

函数参数如表 4-3 所示。

表 4-3 函数参数

参数	描述
sem	释放信号量对象

第5章 移植

SMC-RTOS 本身是一个轻量的内核，仅仅提供线程调度、信号量和软件定时器功能。因此，SMC-RTOS 主要针对小用户或者简单的应用编程者。所以现阶段的 SMC-RTOS 仅针对 Cortex-M3 和 Cortex-M4 内核的移植。与 CPU 相关的代码存放在 cortex-m3.c 或者 cortex-m4.c 文件中。根据自己使用的 CPU 内核去选择自己的开发平台。假如使用的是 cortex-m4 平台，只需要将 cortex-m4.c 文件包含到项目工程里面。其他文件中的代码为硬件无关代码，与移植无关。

SMC-RTOS 文件结构如图 5-1 所示。bsp 目录存放外设初始及应用代码，例如：main.c 文件为 SMC-RTOS 启动代码，board.c 文件存放外设硬件初始化，app.c 存放上层应用程序。config 目录下面存放用户配置文件 smc_config.h 以及用户应用编程需要的 SMC-RTOS 所有 API 声明头文件 smc_rtos.h。libcpu 文件夹存放不同架构 CPU 的移植文件，以内核名称命名，用户使用只需要包含其中一个文件到工程中即可。src 目录为 SMC-RTOS 的系统代码实现，其中 include 目录是系统的头文件存放路径。我们重点关注 smc_cpu.h 文件，该文件中的所有接口函数都是移植需要实现的函数接口。其中 cortex-m3.c 和 cortex-m4.c 文件中的实现就是实现这些接口函数。

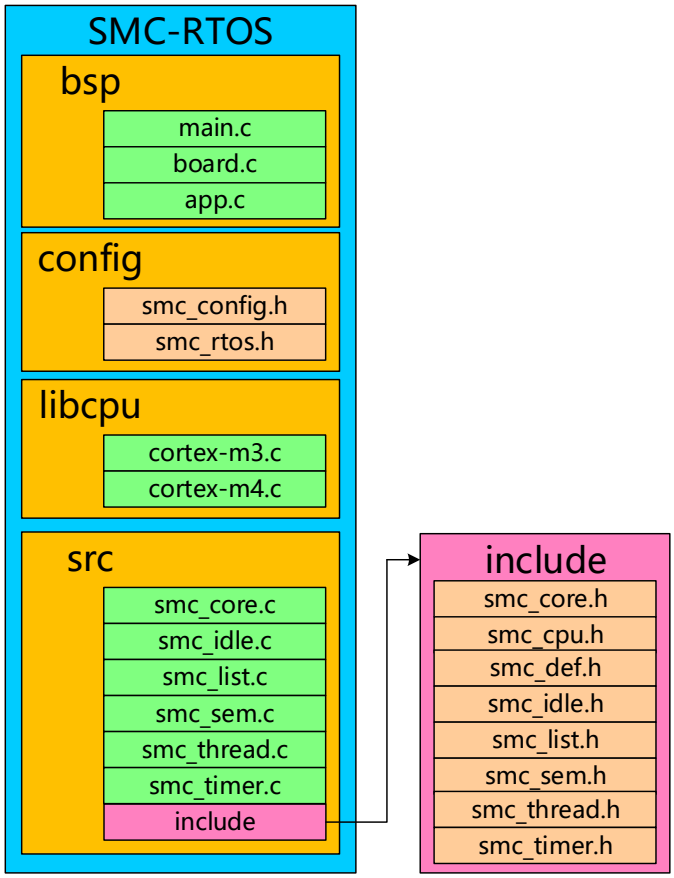

图 5-1SMC-RTOS 文件结构

5.1 用户配置选项

用户配置选项全部采用宏定义实现。

(1)时钟节拍设置

SMC_TICKS_PER_SECOND 定义时钟节拍，一秒钟内包含的时钟节拍个数。一个系统节拍的时间即是系统节拍定时器的定时周期时间。

(2)最大优先级数目

SMC_PERIORITY_MAX 定义了 SMC-RTOS 支持的线程优先级数目，取值范围是 1~32。对应线程优先级 0~31，其中最低优先级分配给空闲线程。

(3)空闲线程栈大小

SMC_IDLE_STACK_SIZE 定义了空闲线程栈的字节大小。

(4)是否使用信号量模块

如所需要使用信号量模块需要宏定义 SMC_USING_SEMAPHORE。