第 8 章基于 STM32F107 的 FreeRTOS 移植

本章以 FreeRTOS 在 STM32F107 上的移植为主要内容，介绍移植的主要工作内容，及移植过程需要注意的事项。

8.1 概述

FreeRTOS 的实现主要由 list.c、queue.c、croutine.c 和 tasks.c 4 个文件组成。list.c 是一个链表的实现，主要供给内核调度器使用；queue.c 是一个队列的实现，支持中断环境和信号量控制；croutine.c 和 task.c 是两种任务的组织实现。协程(英文为 croutine)是采用各任务共享同一个堆栈，使 RAM 的需求进一步缩小，但也正因如此，他的使用受到相对严格的限制。而 task 则是传统的实现，各任务使用各自的堆栈，支持完全的抢占式调度。

FreeRTOS 的主要功能可以归结为以下几点：

- 优先级调度、相同优先级任务的轮转调度，同时可设成可剥夺内核或不可剥夺内核
- 任务可选择是否共享堆栈 (co-routines & tasks)，并且没有任务数限制
- 消息队列，二值信号量，计数信号量，递归互斥体
- 时间管理
- 内存管理

与 UC/OS-II 一样，FreeRTOS 在 STM32 的移植大致由 3 个文件实现，一个.h 文件定义编译器相关的数据类型和中断处理的宏定义；一个.c 文件实现任务的堆栈初始化、系统心跳的管理和任务切换的请求；一个.s 文件实现具体的任务切换。在本次移植中，使用的编译软件为 IAR EWARM 6.3。

8.2 FreeRTOS 移植

8.2.1 portmacro.h 头文件

PORTMACRO.H 头文件主要包括两部分内容。第一部分定义了一系列内核代码中用到的数据类型。FreeRTOS 与 uC/OS-II 一样，并不直接使用 char、int 等这些原生数据类型，而是将其重新定义为一系列以 port 开头的新类型。这样的好处在于，在不同架构的处理器移植时，仅需针对处理器的位宽对这些数据类型做相应的调整，而无需繁琐地从头至尾修改源代码。

第二部分包含了实现 FreeRTOS 移植所需要定义的函数。包括与架构相关的定义、内核调度、临界区管理、任务优化等。

1. 数据类型定义

定义编译器相关的各种数据类型。在 uC/OS-II 的移植代码中，通常采用 typedef 来定义新的

类型，而 FreeRTOS 的作者似乎更喜欢用宏定义 #define。虽然 typedef 和 #define 能够实现相同的功能，但也有细微的区别，typedef 由编译器解释，而非预处理器执行，并且仅限于对数据类型进行定义。尽管如此，typedef 在其受限范围内比 #define 更为灵活。下面是相应的代码片段。

```
#define portCHAR      char
#define portFLOAT     float
#define portDOUBLE    double
#define portLONG      long
#define portSHORT     short
#define portSTACK_TYPE unsigned portLONG
#define portBASE_TYPE long
```

2. 架构相关的定义

定义与处理器或控制器架构相关的宏定义。

条件编译中的代码是针对处理器的字长为 16 位，工程配置时应在 FreeRTOSConfig.h 中将宏定义改为 “#define configUSE_16_BIT_TICKS 1”，若处理器字长为 32 位，宏定义为 “#define configUSE_16_BIT_TICKS 0”。

宏 portSTACK_GROWTH 定义了堆栈的生长方向。对于不同的嵌入式操作系统，堆栈的生长方向定义可能不同，在移植时应予以明确。portSTACK_GROWTH 定义为 1 表示堆栈是正向生长的，-1 为逆向生长的。一般来说堆栈都是倒生的。Cortex-M3 的堆栈增长方向为高地址向低地址增长，因此这里定义为 -1。

portTICK_RATE_MS 表示的是 Tick 间间隔多少 ms，只在应用代码中可能会用到。如使用 vTaskDelay 延时函数可实现任务定时间间隔地执行，调用方法如下。

```
vTaskDelay( 250 / portTICK_RATE_MS );
```

vTaskDelay 传递参数中的 250 表示系统使当前任务处于阻塞态，并维持 250ms。

portBYTE_ALIGNMENT 在 uC/OS-II 是不需要的，FreeRTOS 的代码中在分配任务堆栈空间时实现 SRAM 访问的字节对齐。

```
/* SYSTEM TICK 的长度。如果是 16 位以下的处理器置 1；如果是 32 位处理器，则置 0。*/
#if( configUSE_16_BIT_TICKS == 1 )
    /* 16 位处理器及 16 位以下处理器 */
    typedef unsigned portSHORT portTickType;
    #define portMAX_DELAY ( portTickType ) 0xffff
#else
```

```

/* 32位处理器 */

typedef unsigned portLONG portTickType;
#define portMAX_DELAY ( portTickType ) 0xffffffff
#endif

/* Cortex-M3的堆栈增长方向为高地址向低地址增长 */

#define portSTACK_GROWTH ( -1 )

/*定义心跳时钟周期，表示的是相邻Tick间间隔多少ms */

#define portTICK_RATE_MS((portTickType)1000/configTICK_RATE_HZ)

/* 访问SRAM的字节对齐 */

#define portBYTE_ALIGNMENT 8

```

3. 内核调度函数

portYIELD() 等同于 vPortYieldFromISR() 实现的是任务切换。vPortYieldFromISR() 函数在 PORT.C 文件中实现。相当于 uC/OS-II 中的 OS_TASK_SW()。

```

/* 声明该函数定义在其它文件中，实现强制上下文切换，用在任务环境中调用 */

extern void vPortYieldFromISR( void );
#define portYIELD() vPortYieldFromISR()

/* 强制上下文切换，在中断处理环境中调用 */

#define portEND_SWITCHING_ISR(xSwitchRequired) if(xSwitchRequired)\
vPortYieldFromISR()

```

4. 临界区管理函数

定义临界区的管理函数。vPortSetInterruptMask 和 vPortClearInterruptMask 在 portasm.s 中定义，实现中断屏蔽位的清除或置位。vPortEnterCritical 和 vPortExitCritical 函数在 port.c 文件中定义，实现临界区的进入与退出。而下面代码中的最后两个宏定义则是用于在中断环境的中断屏蔽和开启，即是否允许中断嵌套。

```

/* 声明以下4个函数在其它文件中定义，临界区的进入与退出、中断的允许和关闭 */

extern void vPortEnterCritical( void );

```

```

extern void vPortExitCritical( void );
extern void vPortSetInterruptMask( void );
extern void vPortClearInterruptMask( void );

/* 中断允许和关闭 */

#define portDISABLE_INTERRUPTS() vPortSetInterruptMask()
#define portENABLE_INTERRUPTS() vPortClearInterruptMask()

/* 临界区进入和退出 */

#define portENTER_CRITICAL() vPortEnterCritical()
#define portEXIT_CRITICAL() vPortExitCritical()

/* 用于在中断环境的中断屏蔽和开启 */

#define portSET_INTERRUPT_MASK_FROM_ISR() 0;vPortSetInterruptMask()
#define portCLEAR_INTERRUPT_MASK_FROM_ISR(x) \
vPortClearInterruptMask();(void)x

```

5. 任务优化函数

以下两个宏定义是为了利用某些 C 编译器的扩展功能对任务函数进行更好的优化。`portNOP()` 顾名思义就是对空操作定义了个宏。

```

/* 预留扩展函数，用于对任务函数的优化 */

#define portTASK_FUNCTION_PROTO( vFunction, pvParameters ) void \
vFunction( void *pvParameters )
#define portTASK_FUNCTION( vFunction, pvParameters ) void \
vFunction( void *pvParameters )

/*消耗一个机器周期，用来延时或空操作*/

#define portNOP()

```

8.2.2 port.c 源文件

1. 堆栈初始化

进行堆栈的初始化，使堆栈处于预知的确定状态。下面是堆栈初始化的代码实现，其模拟了一个由中断引起的上下文切换导致的堆栈操作序列。

```

portSTACK_TYPE *pxPortInitialiseStack( portSTACK_TYPE *pxTopOfStack,
                                        pdTASK_CODE pxCode,
                                        void *pvParameters )
{
    /* 计算存储程序状态寄存器xPSR的堆址，用于MCU在进入或退出中断时恢复现场 */
    pxTopOfStack--;

    /* 程序状态寄存器的值保存于堆栈中 */
    *pxTopOfStack = portINITIAL_XPSR;

    pxTopOfStack--;

    /* 任务的入口点 */
    *pxTopOfStack = ( portSTACK_TYPE ) pxCode;

    pxTopOfStack--;
    /* LR */
    *pxTopOfStack = 0;

    /* R12, R3, R2 and R1. */
    pxTopOfStack -= 5;

    /* 任务的参数 */
    *pxTopOfStack = ( portSTACK_TYPE ) pvParameters;

    /* R11, R10, R9, R8, R7, R6, R5 and R4. */
    pxTopOfStack -= 8;
    return pxTopOfStack;
}

```

2. 启动任务调度

```

portBASE_TYPE xPortStartScheduler( void )
{
    /* 让任务切换中断和心跳中断位于最低的优先级，使更高优先级可以抢占MCU */
    *(portNVIC_SYSPRI2) |= portNVIC_PENDSV_PRI;
}

```

```

*(portNVIC_SYSPRI2) |= portNVIC_SYSTICK_PRI;

/* 启动定时器，开始产生系统的心跳时钟，此处中断已被关闭 */
prvSetupTimerInterrupt();

/* 初始化临界区的嵌套的个数，准备启动第一个任务 */
uxCriticalNesting = 0;

/* 启动第一个任务 */
vPortStartFirstTask();

/* 执行到vPortStartFirstTask函数，内核已经开始正常的调度 */
return 0;
}

```

3. 主动释放 MCU 使用权

通过设置中断控制及状态寄存器触发 PendSV 系统服务中断，请求一次上下文切换，中断到来时由汇编函数 xPortPendSVHandler()处理。

```

void vPortYieldFromISR( void )
{
    /*通过设置中断控制及状态寄存器，请求一次上下文切换*/
    *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
}

```

4. 临界区进出

进入临界区时，首先关闭中断，当退出所以嵌套的临界区后再使能中断。这段代码中进出临界区是通过关中断和开中断操作来实现的，相当与 uC/OS-II 中 OS_CRITICAL_METHOD == 1 的情况。不过，通过全局变量 uxCriticalNesting 来记录临界区的嵌套层数以此来实现临界区的嵌套操作。uxCriticalNesting 虽然是全局变量，但是后面可以看到在任务切换时会将 uxCriticalNesting 的值存到当期任务的堆栈中，完成任务切换后从新的任务的堆栈中取出 uxCriticalNesting 的值。每个任务通过这种方式维护各自的临界区嵌套次数 uxCriticalNesting 。实际上，FreeRTOS 和 uC/OS-II 中临界区的概念是相同的，因此在 uC/OS-II 中可用的临界区保护的方法都可以拿到 FreeRTOS 中来。