FREERTOS 在 STM32 的移植

伟研科技 http://www.gzweiyan.com V 1.0

FreeRTOS 作为开源的轻量级实时性操作系统，不仅实现了基本的实时调度、信号量、队列和存储管理，而且在商业应用上不需要授权费。

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。对于 croutine，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：

1) 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
2) 任务可选择是否共享堆栈(co-routines & tasks)，并且没有任务数限制
3) 消息队列，二值信号量，计数信号量，递归互斥体
4) 时间管理
5) 内存管理

与 UC/OSII 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 5.2。

各文件关键部分的实现：

PORTMACRO.H 宏定义部分

1.定义编译器相关的各种数据类型

```
#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

Cortex-M3 的堆栈增长方向为高地址向低地址增长

#define portSTACK_GROWTH (-1)

每毫秒的心跳次数

#define portTICK_RATE_MS ((portTickType) 1000 / configTICK_RATE_HZ)

访问 SRAM 的字节对齐

#define portBYTE_ALIGNMENT 8

3.定义用户主动引起内核调度的 2 个函数

强制上下文切换，用在任务环境中调用

#define portYIELD() vPortYieldFromISR()

强制上下文切换，用在中断处理环境中调用

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

4.定义临界区的管理函数

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允许和关闭

#define portSET_INTERRUPT_MASK_FROM_ISR() 0;vPortSetInterruptMask()

#define portCLEAR_INTERRUPT_MASK_FROM_ISR(x) vPortClearInterruptMask();(void)x

PORT.C C 接口部分

1.堆栈初始化

```
portSTACK_TYPE *pxPortInitialiseStack( portSTACK_TYPE *pxTopOfStack, pdTASK_CODE pxCode, void *pvParameters )
{
	*pxTopOfStack = portINITIAL_XPSR;		/* 程序状态寄存器 */
	pxTopOfStack--;
	*pxTopOfStack = ( portSTACK_TYPE ) pxCode;		/* 任务的入口点 */
	pxTopOfStack--;
	*pxTopOfStack = 0;		/* LR */
	pxTopOfStack -= 5;		/* R12, R3, R2 and R1. */
	*pxTopOfStack = ( portSTACK_TYPE ) pvParameters;	/* 任务的参数 */
	pxTopOfStack -= 8;		/* R11, R10, R9, R8, R7, R6, R5 and R4. */
```

```
        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;

        设置并启动系统的心跳时钟
        prvSetupTimerInterrupt();

        初始化临界区的嵌套的个数
        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**

```
void vPortYieldFromISR( void )
{
        触发 PendSV 系统服务中断，中断到来时由汇编函数 xPortPendSVHandler()处理
        *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
}
```

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

```
void vPortEnterCritical( void )
{
        portDISABLE_INTERRUPTS();
        uxCriticalNesting++;
}
void vPortExitCritical( void )
{
        uxCriticalNesting--;
```

```
        if( uxCriticalNesting == 0 )
        {
                portENABLE_INTERRUPTS();
        }
}
```

4.**心跳时钟处理函数**

```
void xPortSysTickHandler( void )
{
unsigned portLONG ulDummy;

        如果是抢占式调度，首先看一下有没有需要调度的任务
        #if configUSE_PREEMPTION == 1
                *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
        #endif

        ulDummy = portSET_INTERRUPT_MASK_FROM_ISR();
        { 通过 task.c 的心跳处理函数 vTaskIncrementTick()，进行时钟计数和延时任务的处理
                vTaskIncrementTick();
        }
        portCLEAR_INTERRUPT_MASK_FROM_ISR( ulDummy );
}
```

PORTASM.S 汇编处理部分

1.**请求切换任务**

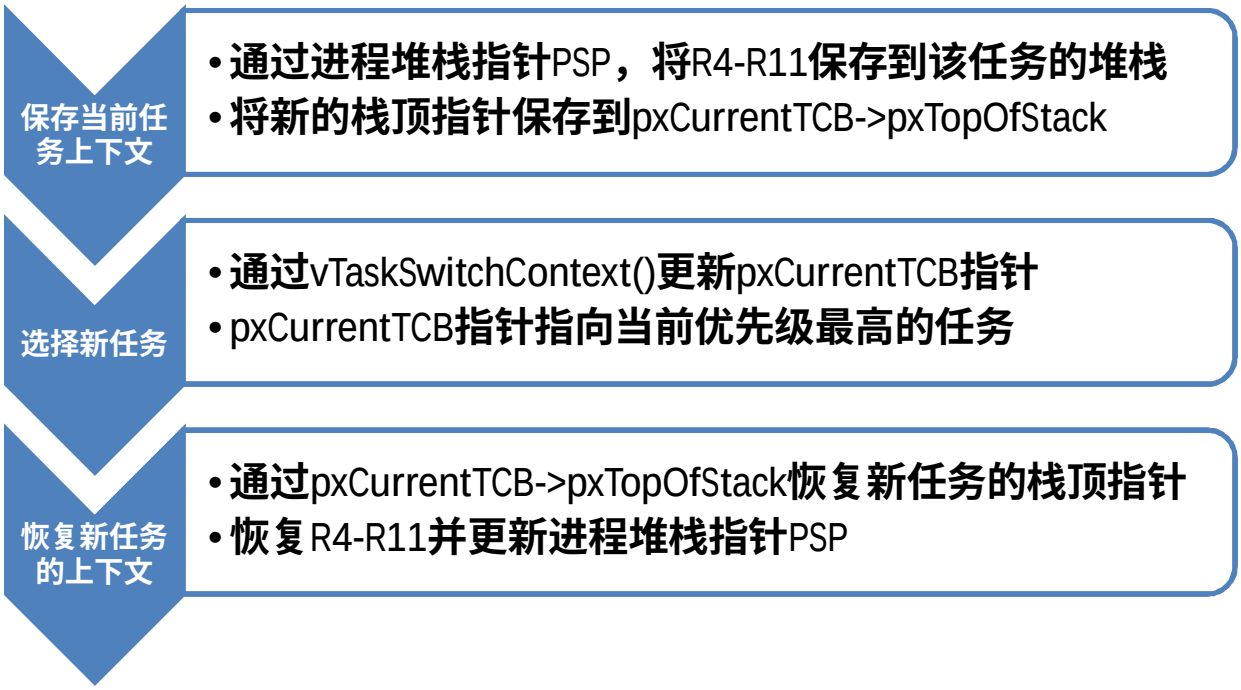
```
xPortPendSVHandler:
        保存当前任务的上下文到其任务控制块
        mrs r0, psp
        ldr     r3, =pxCurrentTCB       获取当前任务的任务控制块指针
        ldr     r2, [r3]

        stmdb r0!, {r4-r11}             保存 R4-R11 到该任务的堆栈
        str r0, [r2]                    将最后的堆栈指针保存到任务控制块的 pxTopOfStack

        stmdb sp!, {r3, r14}
        关闭中断
        mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
        msr basepri, r0
        切换任务的上下文，pxCurrentTCB 已指向新的任务
```

```
        bl vTaskSwitchContext
        mov r0, #0
        msr basepri, r0
        ldmia sp!, {r3, r14}
        恢复新任务的上下文到各寄存器
        ldr r1, [r3]
        ldr r0, [r1]                    /* The first item in pxCurrentTCB is the task top of stack. */
        ldmia r0!, {r4-r11}             /* Pop the registers. */
        msr psp, r0
        bx r14
```

任务切换的示意图如下：



2. **中断允许和关闭的实现，通过** BASEPRI **屏蔽相应优先级的中断源**

```
vPortSetInterruptMask:
        push { r0 }
        mov R0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
        msr BASEPRI, R0
        pop { R0 }

        bx r14

vPortClearInterruptMask:
        PUSH { r0 }
        MOV R0, #0
```

```
MSR BASEPRI, R0
POP     { R0 }

bx r14
```

3.**直接切换任务，用于** vPortStartFirstTask **第一次启动任务时初始化堆栈和各寄存器**

vPortSVCHandler;

```
ldr     r3, =pxCurrentTCB
ldr r1, [r3]
ldr r0, [r1]
ldmia r0!, {r4-r11}
msr psp, r0
mov r0, #0
msr     basepri, r0
orr r14, r14, #13
bx r14
```

4.**启动第一个任务的汇编实现**

vPortStartFirstTask

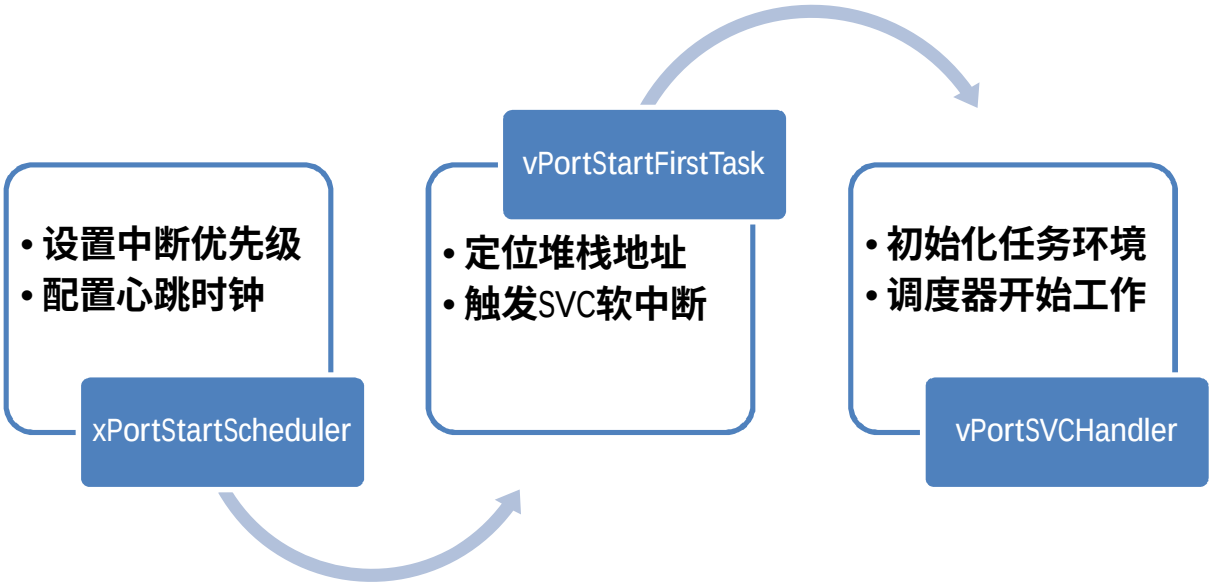
通过中断向量表的定位堆栈的地址

```
ldr r0, =0xE000ED08     向量表偏移量寄存器 (VTOR)
ldr r0, [r0]
ldr r0, [r0]
msr msp, r0             将堆栈地址保存到主堆栈指针 msp 中
```

触发 SVC **软中断，由** vPortSVCHandler()**完成第一个任务的具体切换工作**

```
svc 0
```

FreeRTOS **内核调度器启动的流程如下：**



以上 3 个文件实现了 FreeRTOS 内核调度所需的底层接口，相关代码十分精简。

创建测试任务：

下面创建第一个测试任务，LED 测试

```
int main( void )
{
        设置系统时钟，中断向量表和 LED 使用的 GPIO
        使用 stm32 的固件包提供的初始化函数，具体说明见相关手册
        prvSetupHardware();

        通过 xTaskCreate()创建 4 个 LED 任务 vLEDFlashTask(),
        每个任务根据各自的频率闪烁,分别对应开发板上的 4 个 LED
        vStartLEDFlashTasks( mainFLASH_TASK_PRIORITY );

        创建一个 IDLE 任务后，通过 xPortStartScheduler 启动调度器
        vTaskStartScheduler();

        调度器工作不正常时返回
        return 0;
}

portTASK_FUNCTION()是 FreeRTOS 定义的函数声明，没特殊作用
static portTASK_FUNCTION( vLEDFlashTask, pvParameters )
{
```

```
        ……省略……，具体为计算各 LED 的闪烁频率
        for(;;)
        {
                vTaskDelayUntil( &xLastFlashTime, xFlashRate );
                vParTestToggleLED( uxLED );

                vTaskDelayUntil()的延时时间 xFlashRate，是从上一次的延时时间 xLastFlashTime 算起的，
                相对 vTaskDelay()的直接延时更为精准。
                vTaskDelayUntil( &xLastFlashTime, xFlashRate );
                vParTestToggleLED( uxLED );
        }
}
```

FreeRTOS 的任务创建与 UC/OSII 差异不大，主要参数为任务函数，堆栈大小和任务的优先级。如：

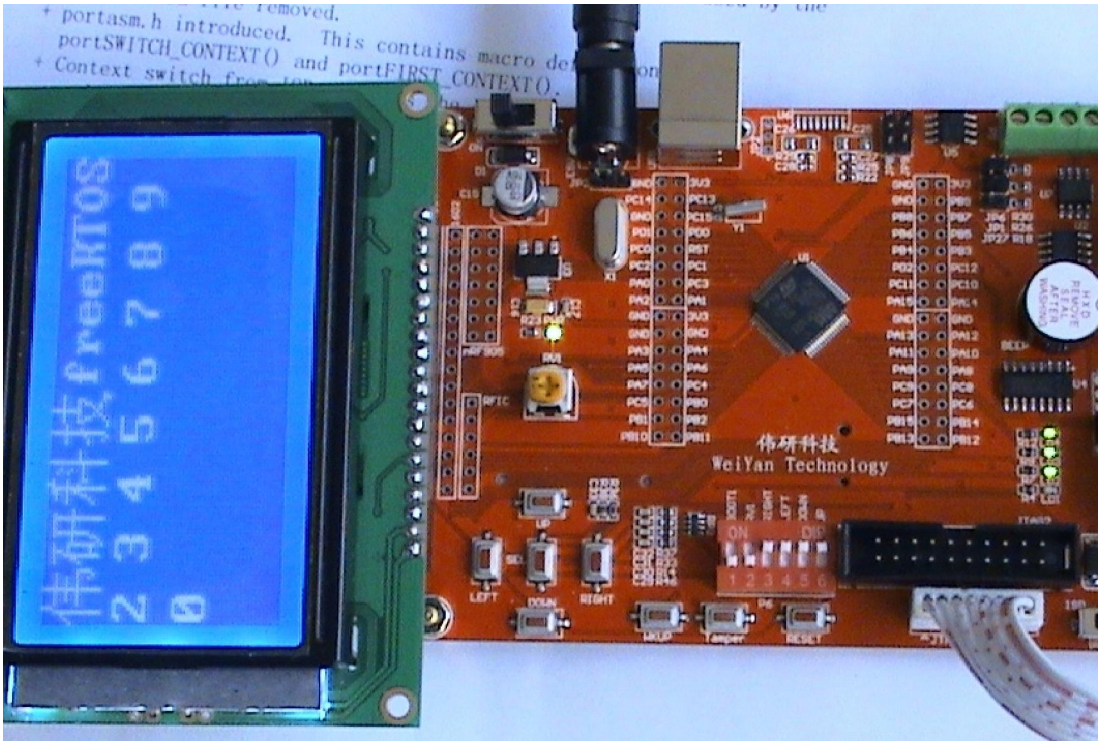
```
xTaskCreate( vLEDFlashTask, ( signed portCHAR * ) "LEDx", ledSTACK_SIZE, NULL, uxPriority, ( xTaskHandle * ) NULL );
```

下面再创建一个 LCD 显示任务，以最低优先级运行：

```
xTaskCreate( vLCDTask, ( signed portCHAR * ) "LCD", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );

void vLCDTask( void *pvParameters )
{
        ……省略……
        for( ;; )
        {
                vTaskDelay(1000);
                printf("%c ", usDisplayChar);
        }
}
```

该任务很简单，每隔 1000 个 ticks(就是 1000ms)，从 LCD 上刷新一个数字。如下图：



至此，FreeRTOS **在** STM32 **上的移植基本完成。与** UC/OSII **相比，**FreeRTOS **精简的实现更适合用来学习实时操作系统的工作原理，对其进行剖析也相对容易。**

接下来，将会移植 CAN，RS485，SD **卡和** USB **等接口到** FreeRTOS，**使其在** STM32 **平台上更加完善。**