天津大学

基于 CortexM 的嵌入式实时操作系统设计与实现

学　　院　智能与计算学部

专　　业　软件工程

年　　级　2015 级

姓　　名　杨兴锋

指导教师　毕重科

职称　副教授

2019 年 5 月 19 日

摘　　要

嵌入式设备网络化、功能复杂化的趋势, 使越来越多的、过去可以用裸奔实现的嵌入式产品, 产生了应用操作系统的需求. 芯片成本的连续下降, 以及 MCU 性能和内存资源的迅速提高, 又为大面积嵌入式实时操作系统 (RTOS) 提供了物质基础. 实时多任务操作系统 (RTOS) 是嵌入式应用软件的基础和开发平台. 目前大多数嵌入式开发还是在单片机上直接进行, 没有 RTOS, 但仍要有一个主程序负责调度各个任务. RTOS 是一段嵌入在目标代码中的程序, 系统复位后首先执行, 相当于用户的主程序, 用户的其他应用程序都建立在 RTOS 之上. 不仅如此,RTOS 还是一个标准的内核, 将 CPU 时间、中断、I/O、定时器等资源都包装起来, 留给用户一个标准的 API(系统调用), 并根据各个任务的优先级, 合理地在不同任务之间分配 CPU 时间. 随着这几年中国物联网的快速崛起, 使得 RTOS 将会被更加广泛的应用. 应用拉动技术, 技术推动应用发展. 中国在物联网 OS 这个点上是有很大机会的, 因为物联网时代目前是中国在引领, 应用需求很大, 会极大地拉动相关技术, 包括 IoT OS 的发展.

本课题旨在 TivaC-LaunchPadGXL123 (ARM Cortex-M4 MCU TM4C123GH6PM) 开发版上实现一个具有抢占式任务调度, 任务同步, 任务间通讯的嵌入式实时操作系统. 以及定义 HAL 接口, 以方便其他开发板的移植.

关键词：　RTOS；　CORTEX-M；　ARM；　任务同步；　任务调度

ABSTRACT

The trend of networked devices and complex functions of embedded devices, Making more and more embedded products that can be implemented in streaking in the past, Produced the need to apply operating systems. The continuous decline in chip cost, And the rapid improvement of MCU performance and memory resources, It also provides a material basis for large-area embedded real-time operating systems (RTOS). Real-time multitasking operating system (RTOS) is the foundation and development platform of embedded application software. At present, most embedded developments are still carried out directly on the microcontroller, without RTOS. But still have a main program responsible for scheduling each task. RTOS is a program embedded in the target code. First executed after system reset, equivalent to the user's main program, The user's other applications are built on top of the RTOS. Not only that, RTOS is still a standard kernel. Wrap up CPU time, interrupts, I/O, timers, etc. Leave the user with a standard API (system call), And according to the priority of each task, Reasonably allocate CPU time between different tasks. With the rapid rise of the Chinese Internet of Things in recent years, Make RTOS will be more widely used. Application of pulling technology, technology to promote application development. China has a great opportunity at the point of the Internet of Things OS. Because the Internet of Things era is currently leading China, the application needs are great. Will greatly stimulate related technologies, including the development of IoT OS.

This project aims to implement an embedded real-time operating system with preemptive task scheduling, task synchronization, and inter-task communication on the TivaC-LaunchPadGXL123 (ARM Cortex-M4 MCU TM4C123GH6PM) development version. It also defines the HAL interface to facilitate other development boards. transplant.

Keywords: RTOS; CORTEX-M; ARM; SwitchContex; Scheduing

目　录

第一章　绪论

1.0.1　研究背景

近年来, 互联网技术的发展, 改变着电子设备、智能终端的形态、功能. 嵌入式设备网络化、功能复杂化的趋势, 使越来越多的、过去可以用裸奔实现的嵌入式产品, 产生了应用操作系统的需求. 芯片成本的连续下降, 以及 MCU 性能和内存资源的迅速提高, 又为大面积应用操作系统提供了物质基础. 回顾裸机时代的开发, 问题也渐渐显现出来.

1.0.2　国内现状

国内目前商用的 RTOS 为数不多, 2015 年沸腾司在华为网络大会发布敏捷网络 3.0 中的一个轻量级物联网操作系统 LiteOS, 和业内众所周知的由阿里开发的 AliOS-Things. 以及最近发展迅速的 RT-Thread.

1.0.3　主要目的

本文旨在细述 RTOS 实现的每一个细节和过程. 为 RTOS 爱好者带来思考.

第二章　　设计

2.0.1　架构

2.0.2　对象容器

第三章　实现

3.1 StartUp

3.1.1 Interrupt Vector Table

首先我们从中断向量表 (既中断服务程序入口地址) 开始, 当中断或异常发生的时候, CPU 自动将 PC 指向一个特定的地址, 这个地址就是中断向量表。Arm 的中断向量表一般位于内存 0x00000000~0x0000001C 处, 结构如下:

表 3-1　Arm 中断向量表

Address	Interrupt Handler
...	...
0x00000040	WWDG_IRQHandler
0x0000003C	SysTick_Handler
0x00000038	PendSV_Handler
0x00000034	Reserved
0x00000030	DebugMon_Handler
0x0000002C	SVC_Handler
0x00000028	Reserved
0x00000024	Reserved
0x00000020	Reserved
0x0000001C	Reserved
0x00000018	UsageFault_Handler
0x00000014	BusFault_Handler
0x00000010	MemManage_Handler
0x0000000C	HardFault_Handler
0x00000008	NMI_Handler
0x00000004	Reset_Handler
0x00000000	Top_Of_Stack

内存中的第一个 4 字节用来初始化 Main Stack Pointer(MSP), 第二个 4 字节是指向 reset handler 函数的指针, 是起始地址, 或者被称为 reset handler 函数的入口点, 用来初始化 Program Counter(PC).

3.1.2 BOOT

当处理器启动时首先读取两个引脚, 包括引脚 boot0 和引脚 boot1, 处理器根据这两个引脚确定引导模式. 然后, 处理器将存储在地址 0x00000000 的值复制到 Main Stack Pointer(MSP), 这一步基本上完成了 Main Stack Pointer(MSP) 的初始化. 接着处理器将存储在地址 0x00000004 的值复制到 Program Counter(PC). 程序计数器始终保存下一个要由处理器执行的操作的内存地址, 因此, 在处理器启动后, 处理器将立即开始执行 reset handler. 通常,reset handler 首先执行一些硬件初始化, 例如数据段和 BSS 段的初始化, 然后 reset handler 调用 main() 函数将控件传递给 main 函数.

表 3-2 Boot Mode

BOOT1	BOOT0	BOOT MODE
x	0	Boot from main flash memory
0	1	Boot from system memory(bootloader)
1	1	Boot from embeded SRAM

大多数 Cortex-M 处理器支持至少三种不同的引导模式, 处理器可以从片内存储器, 系统存储器或片内 SRAM 启动. 存储在系统存储器中的代码称为引导加载程序 (bootloader), 引导加载程序 (bootloader) 通常由芯片制造商提供. 引导加载程序 (bootloader) 可以升级内部闪存内的固件. 所有 STM32 微处理器在只读存储器区域 (ROM) 中都带有预编程的引导加载程序 (bootloader), 该 ROM 区域称为系统存储器.

但是, 有时您需要开发自定义引导加载程序 (bootloader), 例如: 您需要加密固

表 3-3 Memory Map

Size	Address	Memory
0.5G	0xE0000000~0xFFFFFFFF	System
1G	0xA0000000~0xE0000000	External Device
1G	0x60000000~0xA0000000	External RAM
0.5G	0x40000000~0x60000000	Peripheral
0.5G	0x20000000~0x40000000	Internal SRAM
0.5G	0x00000000~0x20000000	Code

件并将其放在 Internet 上, 以便客户可以升级固件. 在这种情况下, 您必须编写自定义引导加载程序 (bootloader) 来解码加密的固件. 表 3-3 是 Arm Cortex-M 处理器的内存映射:

每个存储区的地址范围是固定的. 接下来看一下代码区域 (表 3-4), 代码区域的范围是 0x00000000 到 0x1FFFFFFF. 顶部区域 (0x1FFF000~0x1FFFFFFF) 是保留用于存储引导加载程序的 ROM 区域. 中间区域是片内闪存. 底部区域是可以物理映射到内部闪存, 系统内存的区域. 可以物理映射到内部闪存, 系统内存和内部 SRAM 的区域.

内部闪存, 系统存储器和内部 SRAM 的起始地址也是固定的. 具体来说, 内部闪存的起始地址为 0x08000000, 系统内存的地址从 0x1FFF0000 开始.

表 3-4　Code Area

Size	Address	Memory
todo	0x1FFF77FF~0x1FFFFFFF	Options Bytes
todo	0x1FFF0000~0x1FFF77FF	System memory (bootloader)
todo	0x080X0000~0x1FFF0000	Reserved
todo	0x08000000~0x080X0000	Internal Flash
todo	0x00000000~0x08000000	Alias to flash, system memory or SRAM

现在, 我们回头再看一下引导模式 (表 3-2). 引导模式由引脚 boot1 和引脚 boot0 上的电压决定.

如果引脚 boot0 接地, 则处理器将从内部闪存引导. 如果引脚 boot0 接地, 处理器将物理映射内部闪存到底部区域. 例如, 内存地址 0x08000000 将物理映射到地址 0x00000000. 换句话说, 闪存内容可以从地址 0x00000000 或 0x08000000 访问. 所以, 当处理器启动时, 它总是从内存地址 0x00000000 和 0x00000004 分别获取栈指针 (SP) 和程序计数器 (PC) 的值. 因为内部闪存已经被物理映射到了起始地址 0x00000000, 所以实际上闪存就是是启动内存。

当引脚 boot1 为低电平且引脚引导 0 为高电平时, 系统存储器将物理映射到底部区域. 当处理器从内存地址 0x00000004 获取程序计数器 (PC) 的值时, 处理器实际上是从系统存储器中获取值. 也就是说, 系统存储器被选为引导存储器. 在此启动模式下, 处理器可以重新编程闪存或执行设备固件升级.

当引脚 boot1 和引脚 boot0 都为高电平时, 内部 SRAM 物理映射到底部区域, 内存地址 0x20000000 物理映射到内存地址 0x00000000 因此处理器从 SRAM 引导.

简单一点总结就是 arm 处理器从哪儿引导由 pcb 设计说了算, 然后根据 boot0/1 两个针脚的电压决定把内存的哪块映射到 0 地址区域. 然后处理器取 4 地址的值放到 pc 寄存器开始运行.

3.1.3 StartUp code

那么, 我们从开发者的角度看一下 Start Up 是如何做到的, 以及如何将启动代码放到目标引导内存中:

当链接器 (Linker) 将对象和库文件组合成单个可执行文件时, 链接器脚本 (Linker Script) 提供两种关键类型的操作, 如何对数据和代码段进行操作以及每个部分应放在内存中的位置进行操作. 编程人员可以修改链接描述文件 (Linker Script) 以将代码放在目标引导内存中. 例如这个项目中的链接描述文件:

```
MEMORY {
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 256K
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 32K
}
SECTIONS {
     PROVIDE( _stack_ptr = ORIGIN(RAM) + LENGTH(RAM));
    .text :
    {   _text = .;
        KEEP(*(.vector_table))
        *(.text*)
        *(.rodata*)
        _etext = .; } > FLASH
    .data :
    {   _data = .;
        *(.data*)
        _edata = .; } > RAM AT >FLASH
    .bss :
    {   _bss = .;
        *(.bss*)
        *(COMMON)
        _ebss = .; } > RAM
}
```

从上面的描述文件可以看到代码段, 数据段再内存里面的位置.(此处应有解

释)

然后，我们再看一下如何将值 (Main Stack Pointer(SP) 和 Program Counter(PC)) 存在内存的 0x00000000 地址和 0x00000004 地址, 也就是启动代码, 比如启动汇编文件中放置了这几个值, 例如:

```
Stack_Size      EQU     0x00000400
                AREA  STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE Stack_Size
__initial_sp
...
__Vectors       DCD     __initial_sp    ;Top of Stack
                DCD     Reset_Handler   ;ResetHandler
...
```

代码中 __initial_sp 便是 Stack Pointer(SP),Reset_Handler 便是 Program Counter(PC) 的初始值.

3.2 Interrupt

接下来我们看一下中断是如何在 Arm Cortex-M 微处理器上工作的, 以及为什么需要中断, 假设我们需要开发一个程序, 按下按钮打开红色 LED 灯, 有两种方法可以监视连接到按钮的输入引脚的逻辑状态, 一种是轮询, 另一种是中断, 轮询方法就像是每隔几秒接起电话来查看是不是有电话打进来, 中断方法就像等待电话铃响, 显而易见, 中断方法更高效. 你可以做任何事情, 知道电话响了再把它接起来. 下面是简化了的轮询代码:

```
while(1){
    read_botton_input;
    if(pushed){
        exit;
    }
}
turn_on_LED;
```

在这个循环中, 程序不断读取连接到按钮的引脚, 直到按键被按下, 程序跳出循环, 打开 LED 灯. 轮询方法是一种忙等待的方法, 处理器不停地读取输入直到按钮被按下, 显然, 轮询方法很简单但是低效. 中断方法比轮询更高效, 如果用户按下按钮, 则产生称为中断请求的电信号, 当处理器收到中断请求时, 它会自动暂停正常程序的执行, 并开始执行一个称为中断处理程序的特殊定义函数. 在中断

处理程序完成后, 处理器从暂停的地方重新启动执行常规程序.

然后我们回头看一下 Cortex-M4 的内存映射 (表 3-3), 以及代码区 (表 3-4) 和中断向量表 (表 3-1), 中断向量表保存一个存储器地址数组, 中断表中的每个条目长度为 4 个字节, 每个条目包含一个中断服务程序的起始地址, 简单地说, 中断表包含一个函数指针数组. 为每个中断类型分配一个编号, 称为中断号. 中断号用于索引中断向量表, 当触发中断 x 时,NVIC(Nested Vectored Interrupt Controller) 使用中断号 x 作为索引值来查找中断 x 的相应中断服务程序的地址, 并强制处理器跳转并执行该中断服务程序.

表 3-5 Interrupt Vector Table

Interrupt Number (8 bits)	Memory Address of ISR (32 bits)
1	Interrupt Service Routine for interrupt 1
2	Interrupt Service Routine for interrupt 2
3	Interrupt Service Routine for interrupt 3
4	Interrupt Service Routine for interrupt 4
5	Interrupt Service Routine for interrupt 5
...	...

那么,NVIC(Nested Vectored Interrupt Controller) 控制器如何使用中断号来查找中断向量表. 正如表 3-1, 可以知道第一个四子节是 MSP, 第二个四字节是 PC, 对于 ARM Cortex-M 处理器, 如果中断号为 n, 则指向中断 n 的中断服务程序的指针存储在中断向量表中的如下地址处:

$$Address of pointer = 64 + 4 * n$$

例如对于中断 $EXTI3\_IRQn = 9$, 他的中断服务程序地址便存在地址

$$Address of pointer to EXTI3 ISR = 64 + 4 * 9 = 100 = 0x64$$

这段描述放在内存里面便是表 3-6:

值得注意的是, 中断号可以是负的, 比如 SysTick 的中断号是-1,$SysTick\_IRQn = -1$, 他的中断服务程序地址便存在地址

$$Address of pointer to SysTick ISR = 64 + 4 * (-1) = 60 = 0x3C$$

也就是说 0x00000004 到 0x0000003C 位置所存的中断服务函数地址所对应的中断服务函数的中断号都是负的, 这十五个中断也叫做 SystemExceptions. 再往后就是厂商特定的中断服务, 中断向量表的大小, 在不同处理器芯片之间有差异.

然后我们通过一个例子在看一下 NVIC(Nested Vectored Interrupt Controller)

表 3-6 Interrupt Vector Table Example

Address	Content	Description
0x20000068		MainStack
...	...	...
0x0800030C		void EXIT3_IRQHandler(){...}
...	...	...
0x00000064	0x0800030D	
...	...	...
0x00000008		
0x00000004	0x2000020D	Reset_Handler();Initialize PC
0x00000000	0x20000068	Initialize MSP(main stack pointer)

是如何处理中断的, 假设外部中断 3(EXTI3) 在这时到达, 以及软件已经使能 (Enable) 了 EXTI3 中断, 通过将中断允许寄存器中的相应位置 1 就可以使能中断, 我们在前面定义了 EXTI3 的中断号是 9, 从中断使能寄存器看, 外部中断 3 已经使能. 中断优先级寄存器显示外部中断 3 的优先级设置为 2.

表 3-7 中断寄存器

Interrupt Number	Enable Register	Active Register	Pending Register	Proirity Register
8	0	0	0	8
9 (EXTI3)	1	0	1	5
10	0	0	0	7
11	0	0	0	4
12 (DMA1_Channel2)	0	0	0	3

NVIC(Nested Vectored Interrupt Controller) 首先将 1 写入外部中断 3 的挂起位. 然后 NVIC(Nested Vectored Interrupt Controller) 开始压栈过程, 并压入 8 个寄存器来保护运行环境, 如果使用了浮点单元 (FPU), 那么更多的寄存器会在压栈 (push) 过程中压入. NVIC(Nested Vectored Interrupt Controller) 首先将程序状态寄存器 (xPSR) 压栈, 然后将程序计数器 (Program Counter(PC)) 即 r15 压栈, 然后将 Link 寄存器 (LR) 即 r14 压栈, 然后是 r12,r3,r2,r1, 最后压入 r0. 需要注意的是 Arm Cortex 使用完全下降的栈.

如果将 32 位项目推入栈, 则栈指针 (SP) 将减少 4, NVIC(Nested Vectored Interrupt Controller) 将这八个寄存器压栈之后, 栈大小会增加 32 字节, 并且栈指

表 3-8 Stack

Register	Address
xxx	SP + 0x20
xPSR	SP + 0x1C
r15	SP + 0x18
r14	SP + 0x14
r12	SP + 0x10
r3	SP + 0x0C
r2	SP + 0x08
r1	SP + 0x04
r0	SP

针 (SP) 会减 32. 上面的压栈过程是由 NVIC 控制器自动操作的, 而不是用户代码. 然后 NVIC 查找中断向量表找到外部中断 3(中断号 9) 的起始地址, 接着 NVIC 控制器将中断 9 的状态由挂起状态 (pending) 改为活动状态 (active).

完了之后,NVIC 强制处理器跳转到中断处理程序, 并开始执行终端处理程序. 中断服务程序通过执行 BX LR 来完成他的运行, 指令 BX LR 告诉 NVIC 控制器执行出栈操作. 同时, 中断活动寄存器 (active register) 的活动位 (active bit) 会被清零. 出栈操作会把那 8 个寄存器从栈中弹出 (pop), 因此,NVIC 恢复处理器状态或者叫做运行环境到中断开始前的状态.

然后再看一下出栈操作, 首先 NVIC 弹出 r0, 然后是 r1,r2,r3,r12,LR,PC, 最后弹出程序状态寄存器 xPSR. 出栈操作完成之后, 运行环境也就从栈中被恢复了. 所有寄存器都有其原始值, 就好像中断从未发生过一样. 作为结果, 处理器成功继续执行被外部中断 3 中断的用户程序.

再看一下 NVIC 如何处理多中断, 假设外部中断 EXTI_3(中断号 9) 此时到达, NVIC 首先将那八个寄存器压栈, 然后让处理器执行中断 9(EXTI_3) 的处理程序, 假设另一个中断请求 (DMA1_Channel2) 在中断 9 处理程序结束之后到达, 此外, 这个新的中断比当前的服务的中断具有更高的紧迫性, 从表 3-7 可以看到,(DMA1_Channel2) 的优先级是 3, 而 (EXTI_3) 的是 5, 需要注意的是对于 Arm Cortex 处理器, 更低的优先级值实际上代表更高的紧迫性, 于是,NVIC 不得不响应这个新的中断, 因为新的中断有更高的紧迫性, 然后 NVIC 停止当前的中断服务程序, 执行另一个压栈过程, 再一次, 它把另外 8 个寄存器放到栈里, 如表 (3-9), 需要注意的是两个寄存器集合有不同的值, 前面的是用户程序的寄存器值, 后面的是中断 9(EXTI_3) 的处理程序的寄存器值, 压栈过程完成之后 NVIC 开始执行

新中断的中断处理程序, 也就是说停止当前中断去处理另一个中断, 这个过程也叫做中断抢占 (interrupt preemption).

表 3-9 Stack

Register	Stack
xxx	
xPSR	User Program
r15	
r14	
r12	
r3	
r2	
r1	
r0	
xPSR	Interrupt9(EXTI_3) Service Routine
r15	
r14	
r12	
r3	
r2	
r1	
r0	

中断 12(DMA1_Channel2) 的中断处理函数完成之后, NVIC 又执行出栈过程, 从栈里弹出 (pop) 八个寄存器, 恢复中断 9(EXTI_3) 的中断服务程序的运行环境, 然后 NVIC 继续执行中断 9(EXTI_3) 的服务程序, 中断 9(EXTI_3) 的服务程序执行完成之后,NVIC 执行继续执行出栈操作, 将中断 9(EXTI_3) 的八个寄存器出栈, 恢复用户程序的运行环境, 用户程序恢复执行. 如果一个比当前中断由更低紧迫性的中断请求到达, NVIC 并不会中断当前的中断处理函数, 而是让它处于挂起状态, 继续当前的中断处理函数, 直到当前的中断处理函数执行完成之后, 再执行出栈, 压栈过程再执行这个更低紧迫性的中断的中断处理函数, 那么这个过程中出栈压栈的过程是不需要的, 因为都是用户程序的寄存器. 因此,Arm Cortex-M 部署了一种称为末尾连锁 (Tail-Chaining) 的优化技术, 以减少中断延迟. 通常, 出栈过程和压栈过程都需要 12 个时钟周期 (cycle), 但是末尾连锁 (Tail-Chaining) 省略了出栈压栈过程之后只需要 6 个时钟周期.

3.2.1 SysTick

系统计时器 (System Timer), 也称为 System Tick 或 SysTick, 用于固定的时间间隔生成 SysTick 中断. 首先, 系统计时器可以测量经过的时间, 所以软件可以使用系统计时器来实现延时功能, 其次我们可以利用它实现定期执行一些特殊的任务, 例如我们可以使用系统计时器定期轮询来检查外围设备状态或者定期读取外部输入, 另外, 操作系统依靠系统计时器来实现 CPU 调度以支持多任务处理和提高 CPU 利用率, CPU 调度器定期的从准备队列里面选取一个新的进程来作为下一个执行的进程.

系统计时器是 ARM Cortex 处理器内置的标准硬件组件, 几乎所有的 ARM Cortex 处理器都具有系统计时器组件, 如果使能, 他可以定期产生 SysTick 中断请求, NVIC 会监控并处理所有的中断请求根据它们的优先级, 对于 SysTick 中断,NVIC 强制处理器执行中断服务程序 SysTick_Handler().

系统计时器是一个 24 位递减计数器, 计数器从重载值递减到 0, 计数器递减到 0 后, 系统计时器会复制存储在重载值寄存器里面的值, 然后系统计时器再次开始倒计时, 当计数器从 1 转换为 0 的时候会产生 SysTick 中断请求, 那么 SysTick 中断的间隔便是:

$$SysTickInterruptTimePeriod = (SysTick\_LOAD + 1) * ClockPeriod$$

然后我们看一下 SysTick 控制和状态寄存器 (SysTick_CTRL), 在该寄存器中只有 4 位是有用的, 一个状态位和 3 个控制位, 状态位 (16 位) 是计数器标志位 (COUNTFLAG), 三个控制位分别是时钟源选择位 (Clock Source), SysTick 中断使能位 (TICKINT) 和定时器使能位 (ENABLE). 正如前面所述,24 位计时器从重载值向下降至 0, 当计数器从 1 降到 0 时, 计数标志位 (COUNTFLAG) 设为 1,

系统计时器由四个寄存器控制, 包括控制和状态寄存器, 重载值寄存器 (SysTick_LOAD), 当前值寄存器 (SysTick_VAL) 和校准寄存器 (SysTick_CALIB), 重载寄存器有 32 位, 前八位不使用, 他可以保存 24 位值, 最大值为 0xFFFFFF, 即 16777215, 计数器从重载值向下计数到 0, 将 0 写入重载值寄存器可以禁用 SysTick, 与 TICKINT 无关. 两个连续的 SysTick 中断的间隔是重载值加 1 的时钟源周期倍, 例如两个连续的 SysTick 中断之间需要 100 个时钟周期, 那么重载值就应该是 99.

当前值也是前八位不用, 读取该寄存器的值可以得到当前的计数器的值, 当当前值从 1 变成 0 的时候会产生 SysTick 中断, 写入 SysTick_VAL 会将计数器和 COUNTFLAG 清零, 使计数器在下一个定时器时钟重新加载, 但是不会触发 SysTick 中断, 需要注意的是, 它在复位时有随机值. 因此, 软件应该始终在初始化的时候将它清零, 可以通过将 0 写入当前值寄存器实现.

然后看一下初始化并启用系统计时器的代码:

```
void SysTick_Initialize(uint32_t ticks){
    SysTick->CTRL = 0; // disable systick
    SysTick->LOAD = ticks-1; // set reload register
    /* set the SysTick interrupt priority (highest) */
    NVIC_SetPriority(SysTick_IRQn, 0U);
    SysTick->VAL = 0; // reset the SysTick counter value
    /* select processor clock:1=processor clock;0=external clock */
    SysTick->CTRL |=SysTick_CTRL_CLKSOUCE;
    /* enable SysTick interrupt: 1=enable;0=disable */
    SysTick->CTRL |=SysTick_CTRL_TICKINT;
    /* enable SysTick: 1=enable;0=disable */
    SysTick->CTRL |=SysTick_CTRL_ENABLE;
}
```

关于 SysTick 还需要知道的是重载值的计算, 假设驱动定时器计数器的时钟源的频率为 80MHz, 我们希望每隔 10ms 产生一次 SysTick 中断, 那么重载值就是:

$$\begin{aligned} Reload &= \frac{10ms}{ClcokPeriod} - 1 \\ &= 10ms * ClockFrequency - 1 \\ &= 10ms * 10^{-3} * 80 * 10^{6} - 1 \\ &= 799999 \end{aligned}$$

3.2.2 本章小结

这一章细述了 ArmCortex 的中断, 尤其是我们在 RTOS 开发中会用到的 SysTick 中断和 PendSV 中断. 这些算是为下一章做个铺垫.

3.3 RTOS

首先需要澄清一点, 本文旨在细述 RTOS 实现的每一个细节和过程, 我所说的 RTOS 特指一个 RTOS 的实时内核组件, 负责多任务处理, 具体并不是指: 硬件抽象层 (HAL), 设备驱动程序, 文件系统, 网络或者其他有时候也归于 RTOS 的组件. 我们在裸机程序的时候都知道, 我们的程序一般都是写在一个死循环里, 让 CPU 不停的执行那个循环, 整个程序采用顺序结构, 或者就是事件驱动, 通过 case 事件来执行对应的代码, 本质是一个大的状态机, 所有应用逻辑全部写在死循环里, 如果功能稍微复杂一点, 结果可想而知, 完全无法维护, 其次对于一些延时功

能, 调用 delay 函数, 一旦程序内的一个功能使用 delay, 就得考虑会不会影响其他功能. 开发与维护难度可想而知, 所有的功能逻辑几乎都是串行起来工作的, 这个时候 CPU 就会有很多时间都浪费在了延时函数里, 一直在空转, 导致软件的并发效率非常差.

我们都知道每个任务是一个函数, 每个函数里面是一个死循环, 如下:

```
1  void task_01_function(){
2      while(1){
3          // do something
4      }
5  }
```

事实证明, 在顺序结构下面, 不可能简单的一个接一个的调用这些函数, 因为大码不可能执行出第一个函数, 也就是说后面的函数不可能被执行, 我们的目的就是探索这种执行的可能性, 也就是多任务处理.

3.3.1 多任务与线程切换

我们在前一章说到 SysTick 中断应用场景的时候说到过操作系统依靠系统计时器来实现 CPU 调度以支持多任务处理和提高 CPU 利用率, CPU 调度器定期的从准备队列里面选取一个新的进程来作为下一个执行的进程. 以及在 NVIC 处理中断服务程序的时候关于用户程序运行现场的保存和恢复, 详细的讲述了, 中断的机制, 尤其是中断发生的时候, NVIC 如何将用户程序压栈 (Stacking), 以及中断服务程序结束之后, 出栈的过程, 即压入 8 个寄存器 (xPSR,PC,LR,r12,r3,r2,r1,r0) 来保护现场, 等中断服务程序结束之后, NVIC 依次弹出 r0,r1,r2,r3,LR,PC,xPSR. 然后处理器继续从 PC 执行, 那也就说我们在 SysTick 中断结束的地方打个断点, 然后中断结束之后手动将栈中 PC 的值修改为另一个函数的地址, 就可以实现让 CPU 去处理另一个函数, 那么重复该过程, 就可以让 CPU 在多个任务之间跳转, 这个方法说明在执行多个后台循环之间切换 CPU 是有可能的, 也指出了这种 CPU 上下文切换的一般机制, 即利用终端处理器中已有的中断处理硬件. 同样也说明了单个 CPU 上进行多任务处理的一般概念, 即在不同的后台循环之间切换 CPU, 当然这种方法是非法的, 原因会在后面解释.

RTOS 内核的简单定义是: 通过允许您在单个 CPU 上运行多个后台循环 (称为线程或任务) 来扩展基本前台/后台架构的软件. 对于多进程或者多线程的理解大致是这样, 频繁地将 CPU 上下文从一个线程切换到另一个线程, 以创建一个每个这样的线程都拥有整个 CPU 错觉. 这两个定义都使用了术语“线程”, 但是需要记住的是, 这些线程本质上是来自前/后架构的后台循环.

现在我们回过头来解释为什么更改栈上的 PC 寄存器值是非法的, 以及怎样做才能将上下文从一个线程干净的切换到另一个线程. 为了说明这个问题, 可以举一个例子, 假设我们有两个任务, CPU 先运行任务 1, 在常规中断抢占的情况下,NVIC 可以保存任务 1 的寄存器并恢复相同任务 1 的寄存器, 一切正常, 但是手动修改返回地址位任务 2 时,CPU 可以返回到任务 2, 但是仍然恢复当初为任务 1 保存的寄存器, 这是非法的.

所以, 需要将不同线程的寄存器集分来, 换句话说, 为任务 1 保存的寄存器无法恢复为任务 2, 反之亦然. 这意味着需要为每个线程使用私有的堆栈, 似乎很复杂, 实际上并非如此. 我们可以很容易的将一个堆栈添加到一个线程, 因为它实际上只不过是 RAM 中的一个区域和一个指向该堆栈当前顶部的指针. 在 C 语言中, 这样的存储区可以表示为 uint32_t 类型的数组 (对于 CPU 的 23 位寄存器) 加上堆栈指针.

```
1    uint32_t stack_task_01[40];
2    uint32_t *sp_stack_task_01 = &stack_task_01;
```

这样的话在主程序中就不需要再调用线程函数, 相反, 需要使用伪造的 Cortex-M 中断堆栈帧预填充每个线程的堆栈. 目的是使堆栈看起来像在调用线程函数之前被中断抢占一样, 因此, 需要使用 DataSheet 中 ARM 异常帧布局作为模版.

从堆栈的高内存地址端开始, 因为 ARM 堆栈从高内存地址增长到低内存地址, 此外需要注意的是 ARM CPU 要求 ISR 堆栈帧需要 8 字节对齐, 最后 ARM CPU 使用“Full-Stack”, 这意味着堆栈指针指向最后使用的堆栈条目, 而不是第一个空闲条目. 因此, 要添加新的堆栈条目, 首先将堆栈指针递减到第一个空闲位置, 然后取消引用它以将值写入此位置. 根据 ARM 异常帧布局, 需要填充的第一个值是程序状态 xPSR, 这个只需要设置第 24 位就可以, 该位对应 ARM 的 THUMB 状态, 表示 ARM CPU 使用 THUMB 指令集, 实际上 ARM 只支持 THUMB 指令集. 由于历史原因,xPSR 寄存器必须设置 THUMB 位. 即

```
1 --sp_stack_task_01 = 0x1 << 24;
```

堆栈上的下一个值是 PC(Program Counter), 这是中断的返回地址, 它需要设置为线程函数说的地址. C 语言允许使用与获取变量地址完全相同的 & 运算符来获取函数的地址, 它使用函数的 address-of 运算符来生成一个指向函数的指针, 然后需要将它强制类型转换为 uint32_t.

```
1    --sp_stack_task_01 = &task_01_function;
```

ISR 堆栈帧中的其他寄存器对于正确调用线程函数并不重要, 因为县城不会返回, 但是处于测试目的, 可以使用寄存器编号对应的数字初始化堆栈, 这有助于

识别调试器中的堆栈帧.

```
    --sp_stack_task_01 = 0x0000000EU; /* LR */
    --sp_stack_task_01 = 0x0000000CU; /* R12 */
    --sp_stack_task_01 = 0x00000003U; /* R3 */
    --sp_stack_task_01 = 0x00000002U; /* R2 */
    --sp_stack_task_01 = 0x00000001U; /* R1 */
    --sp_stack_task_01 = 0x00000000U; /* R0 */
```

那么现在在 SysTick 结束时设置断点, 将 CPU SP 寄存器的值修改为 sp_stack_task_01 就可以将 CPU 堆栈切换为私有 task_01 堆栈. 要想将上下文切换为其他的线程, 可以重复上述步骤. 假设需要将上下文切换为另一个线程 task_02. 需要在更改 CPU 中的 SP 寄存器之前, 将 SP 的值从 CPU 复制到 sp_stack_task_01 堆栈指针变量中, 因为这实际上是 task_01 线程的当前堆栈顶部, 因此需要更新堆栈指针切断这个线程, 然后用 task_02 线程的堆栈顶部覆盖 CPU SP 寄存器来执行.

上面说明了为每个线程使用单独的私有堆栈的上下文切换的新方法, 这种方法不再使用混合寄存器, 相反,task_01 线程的寄存器存储在 sp_stack_task_01 堆栈中, 随后从相同的 sp_stack_task_01 堆栈恢复. 所有这些看起来很有希望作为线程上下文切换的方式, 但是, 我们还没有完全走出困境. 剩下的问题是这种上下文切换仍然会破坏某些 CPU 寄存器, 因为在恢复给定线程之前,CPU 状态未正确恢复.

Cortex-M 异常堆栈帧对用于 ARM 应用程序过程调用标准 (AAPCS), 因为它储存允许被函数调用破坏的寄存器, 但是不存储 R4-R11, 所以必须通过函数调用保存. 这适合在中断服务程序 (ISR) 中完成, 因为 ISR 必须在返回到抢占点之前运行完成,

假如 task_01 线程使用了 R7 寄存器, 它并不会保存在 Cortex-M ISR 堆栈帧中. 这时候 ISR 也可以使用 R7, 但是他必须保存并在返回之前恢复, 也就是说如果 ISR 是线程在被抢占之后唯一执行的代码, 它就能正常工作. 但是在我们这种情况下,task_01 线程被抢占后不是返回到 task_01 线程, 而是返回到另一个线程:task_02, 线程 task_02 也可以使用 R7 寄存器, 因为 AAPCS 也有义务在返回时恢复 R7. 但是如果没有执行整个线程函数, 而只是执行了一部分的话, 这段代码不需要符合 AAPCS, 它可以更改 R7 的值, 结果是, 当 task_01 恢复执行时,R7 寄存器可能已经被破坏, 这是一个问题, 当然, 对于其他寄存器 R4~R11 都存在这种情况. 解决方案是在 ISR 结束时, 即将上下文切换之前, 将剩余的 8 个寄存器 R4~R11 保存在线程堆栈上, 然后在 ISR 返回到此线程之前从线程堆栈中恢复这些寄存器.

不妙的是, 这额外的 8 个寄存器会给目前为止所做的手动上下文切换增加不少繁琐的工作, 首先, 需要将附加寄存器 R4~R11 附加到所有线程的堆栈帧中, 其次, 在保存当线程上下文时, 需要在当前 ISR 堆栈顶部保存额外的 8 个寄存器, 此外, 需要从 SP 中减去 0x20 来调整 SP CPU 寄存器的值, 然后将其保存在线程的堆栈指针中. 然后在恢复下一个线程时, 需要将附加寄存器 R11~R4 从线程堆栈恢复到 CPU 寄存器, 最后, 在将线程堆栈指针写入 CPU SP 寄存器之前, 需要给线程堆栈指针加 0x20.

3.3.2 线程初始化

从前面的手动切换线程的过程中可以看到, 每一个线程都需要私有堆栈指针 SP, 以及其他信息, 为了方便对它们进行管理, 以及方便在后续扩展, 我们定一个结构 os_task 来保存它们, 在标准 RTOS 实现中, 与线程相关的数据结构传统意义上称为线程控制快 (TCB), TCB 中的信息大体分为两类: 一类是线程的私有数据, 包括 PC(Program Counter),SP(Statk Pointer), 其他寄存器 (Context) 如 LR,R0-R3,R4-R7. 第二类是这些 TCB 使用的其他数据例如, 调度队列, 锁, 等待列表等. 在我们的 TCB 里面, 暂时需要线程启动函数, 线程启动函数需要私有堆栈的内存和该堆栈的大小, 以及线程在等待时需要放弃 CPU, 所以还需要 timeout. 调度时候的优先级 priority. 线程状态 state, 线程 id 等,

```
typedef struct os_task{
    cpu_stk_t           sp;
    cpu_stk_size_t      stackSize;
    os_task_handler_t   taskHandler;
    os_task_id_t        id;
    os_obj_t            obj;
    task_state_t        state;
    os_time_t           timeout;
    priority_t          priority;
    os_list_t           taskList;
}os_task_t;
```

对于线程启动函数 taskHandler 的, 可以使用 C 语言里面指向函数的指针, 在这里它是一个指向不带参数, 并返回 os_err_t 的函数的指针, 定义如下:

```
typedef os_err_t (*os_task_handler_t)();
```

接下来需要定义 RTOS API, 首先需要定义的第一个 API 是在每个线程的堆栈上构造寄存器上下文的函数, 传统上, 这种 RTOS 服务称之为线程创建或者线程启

动, 我们使用 os_task_ 开始, 这个函数需要以下参数: 一个指向 TCB 的指针, 可以将这个参数命名为 me, 这可以作为一个编码约定.

```
os_err_t os_task_create(os_task_t *me,
                        cpu_char_t *name,
                        priority_t priority,
                        cpu_stk_t stkSto,
                        cpu_stk_size_t stackSize,
                        os_task_handler_t taskHandler);
```

在这个函数中, 需要建立初始堆栈指针, 从中构建堆栈帧, 正如前面提到, 在 Cortex-M 上, 堆栈从高地址向低地址增长, 所以需要从堆栈内存的末尾开始, 前面还提到,Cortex-M 堆栈需要在 8 字节边界对齐, 显然调用函数的用户可能并不会知道这些, 所以假设所提供的堆栈内存的末位是正确对齐是不明智的. 我们可以通过舍入地址来确保正确对齐, 即除以 8, 然后整数乘以 8.

```
uint32_t *sp = (uint32_t *)((((uint32_t)stkSto+stackSize)/8)*8);
```

然后用上一小节中同样的方式填充堆栈帧, 并将 SP 寄存器的值赋值为 taskHandler, 除此之外, 可以填充额外寄存器 R11~R4, 最后在 TCB 中保存栈顶指针. 此外, 还可以添加一点额外的功能, 比如用已知的位模式填充剩余的堆栈, 例如使用 0xDEADBEEF, 这将使得很容易就能从内存中查看堆栈. 方便调试.

```
stk_limit = (uint32_t *)(((((uint32_t)stkSto-1U)/8)+1U)*8);
for (sp = sp - 1U; sp >= stk_limit; --sp) {
    *sp = 0xDEADBEEFU;
}
```

3.3.3 中断优先级与中断触发

初始化完了线程, 那么接下来的功能便是上下文切换, 正如在上面章节中说到, 上下文切换需要在在中断返回时发生, 例如 SysTick. 原则上可以在那儿编写代码, 但是这样的话需要将上下文切换添加到系统的每一个 ISR 中, 这不仅是重复的, 而且会破坏代码的整洁性, 因为对于上下文切换, 不能用标准 C 编码, 而是需要一些特定于 CPU 的汇编代码来构建非常特定的堆栈帧以及操作 CPU 堆栈帧寄存器. 但是,ARM Cortex-M 提供了一种解决方案, 允许仅在一个中断中编写上下文切换, 然后根据需要从其他中断或者甚至从线程代码中有效地触发. 在中断那一章我们通过在系统控制模块内的特殊寄存器中设置了一个位来触发了 SysTick, 那么, 我们可以使用同样的方式来针对另一个名为 PendSV 的异常, 几乎

所有的 Cortex-M 的 RTOS 都将其用于上下文切换, 但是,PendSV 并不是那么特别, 原则上可以使用任何其他异常或中断来进行上下文切换. 我们可以修改内存地址 0xE000ED04, 这是系统控制块中的中断控制和状态寄存器. 可以在 DataSheet 中查看到, 通过设置位数 28(即 0x1 后跟 7 个零) 来触发 PendSV 异常, 将该值写入 ICSR 寄存器以触发 PendSV.

```
1    *(uint32_t volatile *)0xE000ED04 = (1U << 28);
```

但是这个时候我们给 SysTick 添加断点会发现,PendSV 抢占了还处于 Active 状态的 SysTick 中断, 显然, 这不是我们想要的结果, 我们希望 SysTick_Handler 执行完成, 而且只有在 SysTick_Handler 执行完成之后,PendSV_Handler 才可以运行上下文切换, 不过, 幸运的是,Arm Cortex-M 内核允许通过对每个异常相关的可调中断优先级来控制异常和中断之间如何相互抢占. 具体而言,SysTick 和 PendSV 的优先级由地址 0xE000ED20 的 SYSPRI3 寄存器控制, 通过在内存中查看此寄存器, 可以看到其中 SysTick 优先级为 0xE0, 且 PendSV 优先级为 0x0.(需要注意的是更高优先级的数字意味着抢占的优先级更低), 这就是优先级为 0 的 PendSV 抢占优先级为 E0 的 SysTick 的原因. 如果我们翻转它, 那就是给 SysTick 优先级 0 和 PendSV 最低优先级 E0, 就可以得到我们需要的抢占顺序. 这里有一点值得注意的是即使将 FF 写入 PendSV 相关的字节, 该值还是会以 E0 的方式读回, 这是因为 ARM Cortex-M 内核仅在优先级字节的最高位中实现中断优先级. TivaC MCU 仅实现三个中断优先级位. 其他 Cortex-M MCU 可能会实现更多位, 例如 STM32 实现 4 个优先级位, 因此如果您将 FF 写入 ST 芯片, 它将回读为 F0. 具体见[?], 但是, 对于今天, 只需要记住 PendSV 需要具有所有异常和中断的最低中断优先级, 可以通过将 FF 写入 PendSV 的优先级字节来设置. 这涵盖 ARM Cortex-M 内核的所有可能版本.

PendSV 优先级设置需要在系统初始化期间进行, 所以我们把它放在 os_kernel_init() 函数中.

```
1    *(uint32_t volatile *)0xE000ED20 |= (0xFFU << 16);
```

在 RTOS 实现中, 推荐使用 SYSPRI3 寄存器的原始内存地址而不是 CMSIS 接口, 因为在任何特定的 ARM Cortex-M 内核, 例如 Cortex-M0,M3,M4 或者 M7 中, PendSV 优先级都处于相同的地址, 因此该代码将更加通用. 在应用程序级代码中, 通常应避免使用具有最低优先级的中断, 因为要为 PendSV 保留最低级别. 因此, 在 bsp.c 中, 需要将 SysTick 的优先级从最低级别提高到零: 我们使用了特定的 TivaC MCU(TM4C123gh6pm), 因此可以使用 CMSIS 函数 NVIC_SetPriority() 来设置 SysTick 异常的优先级.

```
1    NVIC_SetPriority(SysTick_IRQn, 0U);
```

有了中断优先级, 我们需要一个触发 PendSV 的功能, 因为上下文切换的触发将与下一步调度哪个线程的决定密切相关. 因此, 此函数的名称将为 os_sched()。

要实现此调度服务, 首先需要确定如何跟踪当前线程和下一个要执行的线程. 这可以简单地编写为 os_task_t 对象的两个指针. osTaskCurr 指针将指向当前线程,osTaskNext 将指向要运行的下一个线程. 由于这些指针将在中断中使用, 因此需要 volatile, 需要在星号后面放置“volatile”关键字, 因为需要指针 volatile. 如果在星号之前放置了“volatile”, 将得到一个指向 volatile os_task_t 结构的 non-volatile 指针, 这并不是我们想要的. 回到 os_sched() 函数的实现, 它需要决定如何设置 osTaskNext 指针, 我们先跳过这一步.

```
1    os_task_t * volatile osTaskCurr;
2    os_task_t * volatile osTaskNext;
```

现在, 让我们简单地编写如何触发 PendSV 异常, 但仅限于下一个线程实际上与当前线程不同时. 此时, 与所有 RTOS 服务一样, 应该非常仔细地了解竞争条件 (Race Condition). 这实际上是构建 RTOS 的最困难的方面.

因为使用了 os_sched(), 这就导致产生很多围绕当前和下一个指针的 Reac-Condition 的机会, 所以需要通过禁用中断来阻止它们. 有两种选择: 禁用函数内部的中断, 如下所示.

```
1    __disable_irq();
2    if(osTaskNext != osTaskCurr){
3        *(uint32_t volatile *)0xE000ED04 = (1U << 28);
4    }
5    __enable_irq();
```

或者, 始终在已建立的临界区 (Critical Section) 中调用整个函数.

```
1    __disable_irq();
2    os_sched();
3    __enable_irq();
```

推荐使用第二种方式, 因为事实证明, 当已经禁用了中断时, 通常需要调用调度程序, 因此在 os_sched() 内再次禁用和重新启用它们可能会有问题.

现在, 我们就可以在 SysTick_Handler 的末尾调用调度程序, 但是需要像上面一样放在临界区里.

3.3.4 上下文切换

在前面提到过,PendSV 必然需要使用汇编进行编写, 因为上下文切换需要一些特定于 CPU 的汇编代码来构建非常特定的堆栈帧以及操作 CPU 堆栈帧寄存器. 有一种比较简单的方式是先用 C 编写, 然后从编译器生成的汇编代码中复制, 我使用的是 arm-none-eabi-gcc, 可以直接通过-S 参数生成汇编代码, 这部分可以配置在 Makefile 里, 由于论文篇幅字数限制, 编译构建工具链这一部分没法在此细述, 本文的重点也不在这里, 但是不可否认的是, 对于一个操作系统项目, 这部分也是不可忽视的非常重要的一部分, 工欲善其事, 必先利其器.

PendSV 需要做的第一件事便是禁用中断,

```
1    CPSID I
```

接下来, 需要保存当前线程的堆栈上下文. 但是需要小心, 因为第一次没有线程运行, 并且 osTaskCurr 指针将在复位时为 0. 因此需要在 if 语句中检查它. 如果不是 0 则在 if 中将寄存器 r4 到 r11 压入到当前堆栈, 压完寄存器后, 需要将 SP 寄存器保存到当前线程的私有 sp 数据成员中.

```
1    LDR     r1,osTaskCurr
2    LDR     r1,[r1,#0x00]
3    CBZ     r1,PendSV_restore
4    PUSH    {r4-r11}
5    LDR     r1,osTaskCurr
6    LDR     r1,[r1,#0x00]
7    STR     sp,[r1,#0x00]
```

保存当前线程的上下文后, 需要恢复要运行的下一个线程的上下文. 因此, 将 SP 寄存器设置为 osTaskNext 线程中私有 sp 的值. 当正在更改当前线程时, 将 osTaskCurr 指针设置为 osTaskNext.

```
1  PendSV_restore:
2      LDR     r1,osTaskNext
3      LDR     r1,[r1,#0x00]
4      LDR     sp,[r1,#0x00]
5      LDR     r1,osTaskNext
6      LDR     r1,[r1,#0x00]
7      LDR     r2,osTaskCurr
8      STR     r1,[r2,#0x00]
```

最后, 从新堆栈弹出寄存器 r4 到 r11, 重新使能中断, 然后愉快地返回到下一个线程.

```
    POP     {r4-r11}
    CPSIE   I
    BX      lr
```

这样就完成了上下文切换.

3.3.5 循环线程调度

每个 RTOS 的核心都是调度程序, 调度程序负责管理系统中线程的执行. 到目前为止, 在 os_sched() 函数内运行的下一个线程的调度仍然是手动的. 如果对于特定的已知线程, 我们可以 extern 它们, 然后通过 if 对应的条件来调度它们, 但是显然, 这样是不合理的, 那么我们就需要一个数据结构来让他们可以被优雅的调度, 一些 RTOS 将线程组织成一个链表, 然后由调度程序遍历, 我们使用一个简单暴力的解决方案, 即将 TCB 存储在预先分配的数组 os_task_t[] 中. 在调用 os_task_create() 的时候将线程加入到线程数组 os_task_t[] 中, 所以, 你需要的第一件事是 os_task_t[] 数组, 它的大小将为 32 + 1 个线程. 还需要在变量 osTaskNum 中记住到目前为止已经有多少线程被保存. 最后, 调度程序需要记住 os_task_t[] 数组 osTasks 中的当前索引 osTaskIndex, 它将以循环递增. 每次在 os_task_create() 中启动一个新线程时, 指向该线程的“me”指针存储在 os_task_t[] 数数组中,

```
    os_tasks[osTaskNum] = me;
    ++osTaskNum;
```

os_task_num 计数器将递增到下一个线程. 需要注意的是, 这里要判断一下, 防止溢出.

然后将真正的调度代码写在前面定义的 os_sched() 函数中, 首先需要将当前线程索引 osTaskIndex 增加 1, 为了防止溢出, 还需要在它等于 osTaskNum 的时候重置为 0, 然后将 osNextTask 赋值为线程数组里的当前索引线程, 就可以简单的做到循环调度.

```
    ++osTaskIndex;
    if(osTaskIndex = osTaskNum){
        osTaskIndex = 0U;
    }
    osTaskNext = osTasks[osTaskIndex];
    // trigger PenedSV
```

这种设计的优点是不再需要对应用程序中的线程进行硬编码, RTOS 会在每次创建线程的时候将它注册到调度数组中.

考虑现在的实现, 第一次线程被切入运行, 是发生在 SysTick_Handler 中, 因为在那儿调用了 os_sched() 函数, 而 os_sched() 函数对线程进行了调度, 并出发了 PendSV 异常, 然后在 PendSV 异常中实现了线程上下文切换. 所以接下来需要的是在 OS 添加线程之后, 先将第一个线程切入, 然后再开启 SysTick 中断, 让其调度. 所以需要添加一个新的 API os_run(), 在该函数中配置 SysTick 中断, 并调用 os_sched() 将第一个线程切入:

```
1    os_err_t os_run(void){
2        /* callback to configure and start interrupts */
3        os_on_startup();
4        disable_irq();
5        os_sched();
6        enable_irq();
7    }
```

这样的话就通过定时器, 把调度器放在定时中, 就实现了简单的线程循环调度. 还是一种是时间片轮询调度方法, 是一种比较简单易用的调度方式. 但是这种类型的调度通常不利于实时应用. 所以我们不会再讨论它. 后面会实现一种更适合实时应用的优先级抢占调度方式.

3.3.6 延迟与阻塞

现在可以在用户程序中通过简单的代码就可以初始化并执行多个线程, 回到前面例子中提到的线程 task_02:

```
1  os_task_t task_02;
2  uint32_t stack_task_02[40];
3  void task_02_thread(){
4    while(1){
5      light_red_on();
6      delay_block(1000);
7      light_red_off();
8      delay_block(1000);
9    }
10 }
```

这是一个让开发板上红色的 led 亮一秒, 灭一秒的程序, 非常简单. 但是, 线程仍在 delay_block 数内使用原始轮询来做到延时. delay_block 内部的愚蠢轮询是一种非常低效的 CPU 周期浪费, 但是怎么能消除这种浪费呢? 完成前面的工作之后, 我们有了一个全新的工具: 上下文切换. 使用此工具, 就可以用一种完全不同方式处理 delay() 函数. 在延迟开始时, 可以将上下文切换到其他线程, 然后在延迟结束时将上下文切换回它, 而不是启动循环轮询. 在这两个上下文切换之间, 线程将非常有效地阻塞, 根本不消耗 CPU 周期.

从线程的角度来看, 阻塞延迟与轮询延迟没有任何不同. 无论哪种方式, 线程都是简单地调用 delay() 函数, 该函数在延迟过去之前不会返回. 但是从整个系统的角度来看, 这会产生重大影响, 因为两个上下文切换之间的 CPU 周期可供其他实际有事可做的线程使用. 从目前为止的描述中可以清楚地看出, 这种阻塞延迟实现必须成为 RTOS 的一部分, 作为 RTOS 的一部分, 我们将被这个函数称为 os_delay() 函数.

运行轮询 delay_block() 函数的线程一直参与循环调度, 这意味着它会被切入切出. 相反, 处于阻塞状态的线程永远不应该被调度. 它只是不准备运行. 这意味着 RTOS 线程有一个新属性, 它告诉调度程序线程是否就绪. 这个属性便是前面 TCB 里面的 state, 它是个枚举变量, 枚举如下:

```
1    typedef enum task_state{
2        OS_STATE_DORMANT        =   1,
3        OS_STATE_READY          =   2,
4        OS_STATE_BLOCKED        =   3,
5        OS_STATE_RUNNING        =   5,
6        OS_STATE_PENDING        =   4,
7        OS_STATE_INTERRUPTED    =   4
8    }task_state_t;
```

首次创建线程时, 意味着为其分配了 os_task_t 对象和堆栈, 它将变为休眠 OS_STATE_DORMANT 状态, 在这种状态下, 除非在其上调用了 os_task_create() 函数, 否则它无法执行任何操作. 在该调用之后, 一个线程看起来就像是被中断抢占一样, 所以它转换到生命周期的阶段, 为 OS_STATE_INTERRUPTED. 最后, 当线程被调度为运行时, 它将转换为 OS_STATE_RUNNING 状态. 过了一会儿, 线程被调度出来并且另一个线程被调度, 此时原始线程再次被抢占, 状态为 OS_STATE_PENDING. 注意, 在单 CPU 系统中, 一次只能有一个线程处于运行状态. 但是现在, 一个正在运行的线程可以调用 os_delay() 函数, 此时它会被阻塞, 这意味着不准备运行. 状态将转换为 OS_STATE_BLOCKED.

现在考虑一下延迟结束时线程从阻塞状态转换出来的情况, 显然, 这需要由 RTOS 集中管理, 因为阻塞的线程无法执行任何操作, 也就是它无法解除阻塞. 这个中央 RTOS 服务需要定期通过 SysTick 中断激活, 因此称为 os_tick(), 每次激活时,os_tick() 都需要更新各个线程的延迟, 并且需要解除延迟已经过去的线程的阻塞状态.

有趣的是, 线程状态在 Blocked–>Preempted<—->Running–>Blcoked 之间转换. 这也就是说如果线程在处于阻塞状态时未处于就绪状态, 则它必须在“抢占状态”和“正在运行”状态下处于“就绪”状态. 事实上, 当处于 OS_STATE_READY 状态时, 这意味着在 Preempted 或 Running 中, 线程已准备好并愿意运行. 除非有时需要进行抢占以与其他已准备好并愿意运行的线程共享 CPU. 这就有了另一个奇怪的结果. 也就是说, 当前阻塞线程的系统是不完整的, 并且必然需要一个始终准备好运行且不能阻塞的特殊线程.

要看到这一点, 可以考虑具有两个线程的系统, 例如 os_task_01 和 os_task_02. 只要它们都处于 Ready 超级状态, 调度程序就可以运行其中一个. 当其中一个线程进入阻塞状态时, 调度程序仍然可以运行另一个线程. 但是如果两个线程在某个时候都被阻塞怎么办? CPU 仍然必须一次只运行一个线程, 但现在没有一个线程现就绪. 解决方案是添加另一个特殊线程, 使得在当没有其他线程就绪时调度程序依然可以运行. 系统中的这种情况称为“空闲”状态, 因此特殊线程称为“空闲”线程. 不允许此“空闲”线程阻塞, 因此其生命周期不具有阻塞状态.

回到实现,“空闲”线程 osTaskIdle 与 os_task_01 一样, osTaskIdle 的线程里面是一个无限循环, 它会调用一个回调函数 task_idle_thread(),

```
1    os_err_t os_idle(void){
2        while(1){
3            task_idle_thread();
4        }
5    }
```

这样的话应用程序级代码将能够在其中做一些处理. 就像任何其他线程一样, 需要启动空闲线程. 最方便的地方是 os_init(). 现在, 有趣的部分是如何为空闲线程提供堆栈. 一种选择是在 RTOS 中预先分配它, 但是在这个级别你不知道 os_onIdle() 回调将使用多少堆栈. 因此, 最好简单地将空闲堆栈分配推迟到应用程序, 就像在 os_task_create() 函数中完成一样. 现在, 有趣的部分是如何为空闲线程提供堆栈. 一种选择是在 RTOS 中预先分配它, 但是在这个级别你不知道 os_onIdle() 回调将使用多少堆栈. 因此, 最好简单地将空闲堆栈分配推迟到应用程序, 就像在 os_task_create() 函数中一样.

```
os_err_t os_init(cpu_stk_t stkSto, cpu_stk_size_t stackSize){
    // ...
    os_err_t isOsIdleTaskInit = os_task_create(
        &osIdleTask,
        "taskIdle",
        0,
        stkSto,
        stackSize,
        &os_idle
    );
}
```

再来看系统计时器如何更新各个线程的延迟. SysTick_Handler 将调用 os_tick() 函数, 该函数将减少所有非零超时计数器. 当这些向下计数器中的任何一个达到零时, 相应的线程将被解除阻塞并被调度. 正如前面 TCB 所看到的, 在每个线程中有一个单独的超时计数器 timeout 将用来延迟运行. 并且每个线程中的 timeout 相互独立.

3.3.7 优先级线程调度

调度程序的另一种主要方式是抢先式调度, 这是最常见的 RTOS 调度程序类型. 使用抢占式调度程序的话, 正在运行的线程将一直持续到它完成, 或者优先级较高的线程就绪 (在这种情况下, 优先级较高的线程抢占优先级较低的线程), 又或者线程在等待资源 (例如任务调用 sleep()) 时放弃处理器. TI-RTOS 和 FreeRTOS 都具有抢占式调度程序.

回到实现, 对于线程状态 state, 在每一个线程中放置一个是合乎逻辑的事, 但是对于代码的效率, 它并不是最佳的. 相反, 事实证明, 在 32 位位掩码 osReadySet 中将准备运行的标志组合在一起更有效. 此 osReadySet 位掩码将与 os_task_t [] 数组 osTasks 一起使用, osReadySet 位掩码中的每个位对应于 os_task_t [] 数组 osTasks 中的一个线程, 不包含空闲线程, 因为它始终准备运行.

空闲线程始终位于索引零处, 因为它已从 os_init() 启动, 这确保它是 os_task_t [] 数组 osTasks 中的第一个线程. 因此, 跳过索引零意味着 osReadySet 中的位简单地相对于索引移位到 os_task_t [] 数组 osTasks. 例如, 索引 1 处的线程对应于位 0, 索引 2 处的线程对应于位 1, 索引 n 处的线程对应于位 n-1, 并且最后索引 32 处的弹性线程对应于 osReadySet 位掩码中的最后位数 31.

osReadySet 位掩码的各种值对应于以下情况：位 0,1 和 n-1 设置, 表示线程

1,2 和 n 准备好运行. 只有位 1 集表示线程 2 准备运行. 最后, 零位表示系统的空闲状态. 通过简单地在单个指令中将 osReadySet 与零进行比较, 可以非常有效地在代码中检查这种情况. 首先, 通过将 osReadySet 与零进行比较, 快速检查空闲状态, 在这种情况下, 将 osTaskIndex 设置为空闲线程的索引, 即零. 否则, 如果 osReadySet 不为零, 则表示有一些可立即运行的线程. 但是现在不能简单地选择下一个线程索引, 因为线程可能还没准备好运行. 相反, 需要继续以循环方式进行, 直到找到准备运行的非空闲线程. 要检查线程 N 是否准备好运行, 需要用一个仅设置了第 N 位的位掩码和 osReadySet 进行按位与运算. 如果结果为零, 则需要继续, 因为这意味着该位未设置, 因此相应的线程尚未准备好运行. 此外, 需要小心通过跳过索引 0 从循环调度中排除空闲线程. 所以, 这是最终的算法. 以循环方式递增 osTaskIndex, 但是需要在它等于 osTaskNum 的时候重置为 1 而不是零来跳过空闲线程. 只要 osReadySet 指定线程未准备好运行, 就会继续前进.

3.3.8 Semphore

3.3.9 Mutex

3.3.10 MQ

第四章　　期望与总结

需要注意的是, 以上文章提到的代码部分涉及到代码的目录结构, 项目架构, 构建工具链等, 可能无法简单的从文章中简单的了解, 具体可见笔者 GitHub:https://github.com/CenoOS/CenoOS-IOT. 文章部分内容做了简化, 比如线程状态转换, 调度对列, 并未在文中详细道出, 而是用简化的 32 比特位来代替调度对列.

外文资料

Here follows the English paper.

...

中文译文

这里就是外文资料的中文翻译。

...

致　谢

我就做了三件微小的事情，谢谢大家。