摘 要

本次综合课程设计 II 紧密延续综合课程设计 I 的内容的前提下，在自己设计并搭建的四轴飞行器硬件平台上，移植成熟的 uC/OS II 嵌入式操作系统。并且合理划分业务层任务，让在操作系统管理下对各任务进行合理调度，提升 MCU 处理效率，增强实时性。

此外，本学期还深入 Cortex-M4 内核，结合 ARM 处理器汇编知识，分析了微处理器的启动过程，了解了堆、栈、中断向量表等概念并进行实践。同时，我们针对操作系统的事件驱动机制和时间驱动机制进行了研究，实践了裸板汇编中断实验，并且对中断响应的全过程进行了全方位的了解；对 M4 时钟树进行了学习，尤其是对有 SysTick 驱动的操作系统进行了实践，完成了自定义时钟的初始化。

在以上基础上，我们通过自主实现上下文切换代码、进出临界区代码、堆栈初始化代码、时钟驱动代码等，让 uC/OS II 操作系统能够在 STM32F4 处理器上正常运行并执行业务。面向四轴飞行器未来需要完成的业务，合理的划分了数据处理、数据传输、数据显示、电机控制四个任务以及重构了接收机中断和串口中断两个服务。通过在代码中插入 SystemView 代码跟踪工具，能够连接上位机以直观的图形化方式观察操作系统运行情况，帮助分析。

以上本学期基本了实现四轴飞行器硬件子系统与软件子系统的集成。未来将对四轴飞行器飞行控制原理进行实践，最终按时完成课设任务。

关键词：四轴飞行器，STM32，Cortex-M4 内核，uC/OS-II，软件子系统

目 录

说明:

1、报告要求 8000 字以上。

2、本模板仅为基本参考，请各位同学根据个人情况进行目录结构扩展。

3、报告正文必须双面打印。

第一章 针对复杂工程问题的方案设计与实现

1.1 针对复杂工程问题的方案设计与实现

一、 阶段性目标分析

本小组的四轴飞行器方案设计思路可以分为以下几个部分，各硬件模块功能，操作系统统筹各模块，与飞行控制算法。

上学期本小组完成了第一部分各硬件模块功能的实现，本学期完成了移植操作系统，在操作系统的调度下协调各模块。我们先回顾以下第一学期完成的成果:

- 四轴飞行器机械部件搭建
- PCB 转接板绘制打样与焊接
- GY-86 姿态数据处理并进行硬件姿态解算
- WIFI 模块完成上位机通信
- 捕获遥控器接收机产生的 PWM 信号
- STM32 输出 PWM 波通过电调控制电机转速
- 飞行器各部件工作状态由 OLED 屏幕显示
- 一体化集成式飞控 PCB 版绘制

本学期我们将在各硬件模块能顺利工作的基础上让多个模块按照一定逻辑聚合成任务，并且加入嵌入式实时操作系统，使飞行器系统能够根据任务的重要性和时间紧迫性有逻辑，有次序的稳定运行。

本小组还会使用任务级调试工具观察每个任务的运行情况，找出问题，进行优化改进。

二、 操作系统对比与选择

本学期的主要工作就是实现移植嵌入式操作系统。那么为什么要在四轴飞行器上实现操作系统呢？首先需要明确操作系统是什么，能做什么。

操作系统是管理计算机硬件与软件资源的计算机程序。操作系统需要处理如决定系统资源供需的优先次序、控制输入设备与输出设备与管理文件系统等基本事

务。操作系统可以统筹各资源如 CPU。嵌入式操作系统则统筹的是 MCU。

四轴飞行器有多个任务需要运行，但 MCU 一次只能运行一个任务，即一个线程。所以如何使 MCU 能够按照不同任务的需要来正确执行则需要一个管理程序，操作系统即是这样的管理程序，它能依照不同任务的重要性，周期，优先级来分配 MCU 资源，让每个任务都能得到满足，从而让四轴飞行器稳定飞行。中断和轮询都无法达到这样的效果，只有使用操作系统才能让系统有序稳定运行。

本小组采用 uC/OS II 操作系统作为四轴飞行器运行的操作系统。uC/OS II 有诸多优点，它包含了任务调度，任务管理，时间管理，内存管理和任务间的通信和同步等功能。同时它具有稳定，简单，移植性强，参考文献多等特点，对于我们这些操作系统初学者非常友好，适合作为初学者的上手系统。所以本小组采用 uC/OS II。

有了操作系统的支持，还需要对操作系统进行性能评估，性能评估包括每个任务运行时间，切换任务开销，截至时间能否保证等指标。这需要专门的操作系统评估工具。

本小组采用一款叫 Systemview 的操作系统监视分析工具，它可以观察各个任务的活动，比如运行时间，运行次数，开始时间，停止时间等，帮助我们分析优化系统好坏。

三、 软件子系统设计

a) uC/OS II 架构设计

i. uC/OS II 设计任务

在对 uC/OS II 进行设计的时候，我们首先需要学习 uC/OS II 内核的结构和功能，并且能够针对 STM32 开发板进行合理的设置与移植。对其任务调度部分、事件驱动部分、时间驱动部分、启动代码部分等进行有必要的重写以适应 STM32 的运行环境。然后，利用 uC/OS II 操作系统提供的系统 API，对原有功能进行重构，合理划分任务，实现软硬系统集成。

ii. uC/OS II 架构设计

在对 uC/OS II 进行结构梳理后，我们整理出如下的 uC/OS II 结构图。

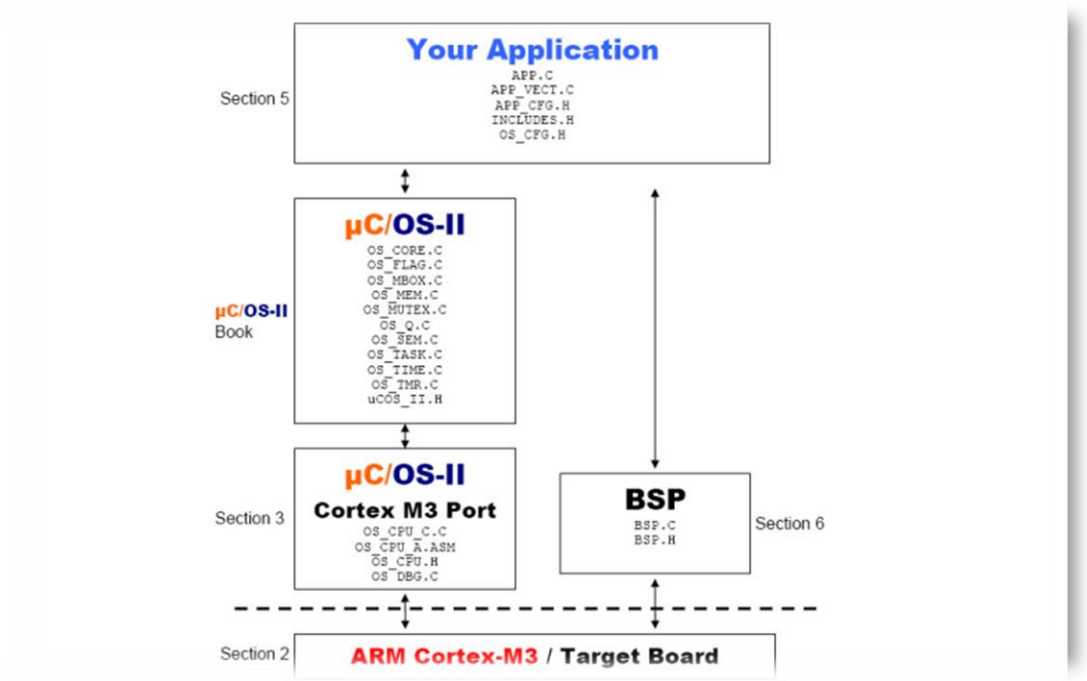


图 1-1 uC/OS II 功能结构

iii. uC/OS II 源代码内核各部分功能简介

- CORE（一般情况不用修改）
  - OS_CORE.c：OS 内核核心初始化，任务调度初始化、Idel 任务、event 等
  - OS_TASK.c：任务的创建、删除、改变 TCB 内容等
  - OS_TIME.c：系统延时等
  - OS_FLAG.c：FLAG 实现，用于任务同步
  - OS_MEM.c：内存管理相关
  - OS_TMR.c：定时器设置相关
  - OS_MUTEX.c：互斥量功能接口实现
  - OS_SEM.c：信号量功能接口实现
  - OS_MBOX.c：邮箱功能接口实现
  - OS_Q.c：队列功能接口实现
  - uCOS_II.h：各种数据结构的定义（如任务、event、链表、信号量等），函数声明
- PORT（根据处理器不同需要调整实现正确移植）
  - OS_CPU_C.c：钩子函数，任务栈初始化、Systick
  - OS_CPU_A.asm：汇编实现的上下文切换核心代码
- CONFIG

- includes.h：
- os_cfg.h：ucOS 功能裁剪配置

- BSP（板级支持包）
  - startup_stm32f401xx.s：stm32 启动汇编代码
  - system_stm32f4xx.c：stm32 系统默认初始化代码

b) OS 任务调度设计

i. M4 寄存器组织

类似与 ARM9 的寄存器，Cortex-M3/M4 也拥有 R0-R15 这些通用寄存器(其中 M4 比 M3 多了一个浮点单元 FPU，此处不展开讲解)；其中，R13 为堆栈指针，在 CM3/CM4 处理器内核中共有两个堆栈，因此便有两个堆栈指针(MSP & PSP)——对比 ARM9 系列的 7 种工作模式来看。MSP 为主堆栈指针，一般在 Handler 模式下使用，由 OS 内核、异常服务程序以及需要特权访问的应用程序代码来使用； 而 PSP 主要用于线程模，用于简单的应用。

设置两个堆栈可能的原因为：为了在进行模式转换的时候，减少堆栈的保存工作；同时也可以为不同权限的工作模式设置不同的堆栈。

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T		GE*	ICI/IT		异常编号				

*GE在Cortex-M4等ARMv7E-M处理器中存在，在Cortex-M3处理器中则不可用。

图 1-2 M4 内核寄存器组织

ii. PendSV 异常

在任务上下文切换的汇编程序中，我们可见 OSCtxSw 的代码只有短短几行，因为它只是触发了一个 PendSV 中断，具体的切换过程其实是在 PendSV 服务函数里面进行的，因此我们会对 PendSV 异常进行简单的讲解。

PendSV，即可悬起的系统调用，OS 可以利用它缓期执行一个异常，直到其它重要的任务完成后才执行操作。触发 PendSV 只需往 NVIC 的 PendSV 悬起寄存器的第 28 为置 1 即可；其典型用于上下文切换。

iii. 中断状态寄存器及其相关定义

在 os_cpu_a.asm 中，有以下两个定义：

```
NVIC_INT_CTRL      EQU      0xE000ED04    ; 中断控制寄存器
NVIC_PENDSVSET     EQU      0x10000000    ; 触发软件中断的值.
```

其中，0xE000ED04 为中断状态寄存器 ICSR(Interrupt Control State Register)，定义为 NVIC_INT_CTRL；

易得 NVIC_PENDSVSET = 0x10000000 ，即第 28 位为 1，其余为 0。

至于为什么这样设置，请见权威手册 8-18 (P162)。

iv. 任务调度

uC/OS-II 是基于任务优先级抢占式任务调度法的，就是内核在管理调度时，调用任务切换函数 OSSched()，在该函数中将此时已处于就绪状态并且为最高优先级的任务的保存于其栈中的相应信息压入 cpu 寄存器中（软中断完成），然后 cpu 开始运行该任务的代码。

内核是何时进行任务调度的呢？uC/OS-II 是可被剥夺资源的内核（高优先级可强行占有低优先级正在使用的资源），内核实时检测是否需要调度的方法是设置调度点，即在调度点中检测是否有更高优先级任务需要调度。 OSTimeDly 这个延时函数里边有一条代码 OSSched()，所以 OSTimeDly 作用就是将此时正在运行的函数挂起（保存任务控制块 OS_TCB 中的相应信息）（任务控制块 OS_TCB 是系统分配给每个任务的信息存储单元），然后调用函数 OSSched()进行任务切换，进而执行就绪的最高优先级任务。此刻，我们了解到 uCOS-II 的任务切换是在执行的任务中调用延时函数 OSTimeDly()进行的。

另一方面，OS 会周期性调用时钟节拍中断服务函数 OSTickISR()进行任务延时计数 OSTimTick()，延时到时调用任务切换函数 OSSched(),进行任务切换。

综上，任务切换有两种途径——时钟节拍中断服务函数 OSTickISR()进行切换，任务中调用时间延迟函数 OSTimeDly()进行切换。从先后顺序来说，应该是 OSTimeDly()先发生（设置 OSTCBDly），才会有时钟节拍中断函数 OSTickISR()进行切换的动作发生。

第三个调度点就是 OSINTEXIT（）在退出中断时也进行任务调度。

调度首先要做的就是找到当前最高优先级的任务并运行它，在 uC/OS-II 中，我

们在任务就绪表中找到最高优先级任务标识（即它的优先级），进而获得该任务的依据——任务控制块。

因为找到最高优先级别并不难，所以调度器 OSSched()的算法也简单。如下:

```
y = OSUnMapTbl[OSRdyGrp];
OSPrioHighRdy = (INT8U)((y<<3) + OSUnMapTbl[OSRdyTbl[y]]);
```

通过上面两行代码将当前最高优先级的任务的优先级存放在 OSPrioHighRdy 变量中。然后通过此变量从存放任务控制块指针的数组 OSTCBPrioTbl[]中获得该任务的任务控制块指针，并存放在指针变量 OSTCBHighRdy 中。代码如下:

```
OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
```

只要获得了最高就绪任务的任务控制块指针，再加上存放在指针变量 OSTCBCur 中的当前运行任务的任务控制块，就可以进行任务切换的工作了。

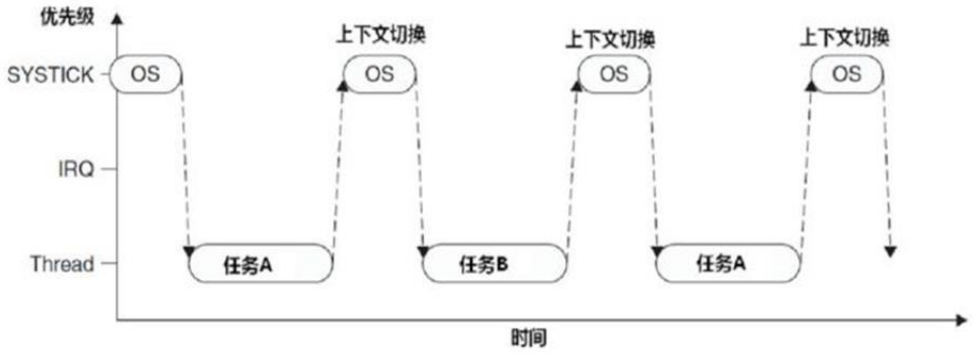


图 1-3 上下文切换示意图

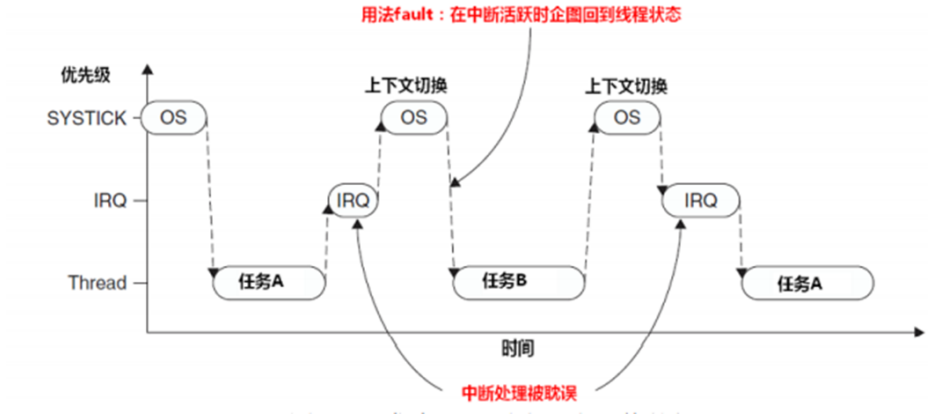


图 1-4 带中断的上下文切换示意图

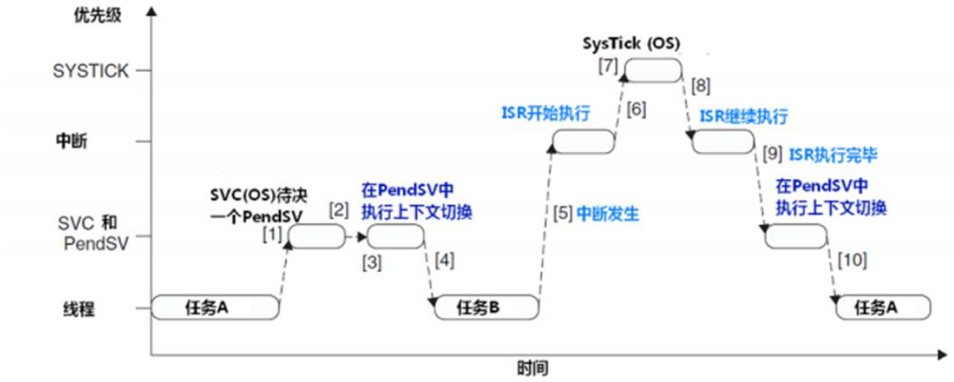


图 1-5 带中断和 PendSV 的上下文切换

c) OS 事件驱动机制设计

i. STM32（M4 内核）中断基本原理

M4 内核中跟中断相关的部分主要是 NVIC（嵌套向量中断控制器）管理模块，它负责控制来自内核内部的异常和来自外部的中断。NVIC 具有以下特性：

- 支持最多 240 个中断输入、不可屏蔽中断 NMI 输入和多个系统异常。每个中断（除 NMI）外，都可以被单独使能或禁止。
- 中断和多个系统异常具有可编程的优先级，对于 M3 和 M4，优先级可以在运行是动态修改。
- 嵌套中断/异常以及中断/异常按照优先级自动处理。
- 向量中断/异常。这就意味着处理器会自动取出中断/异常向量，无需软件确定产生的是那个中断/异常。
- 向量表可以重定位在存储器的多个区域。
- 低中断等待。对于具有零状态的存储器系统，中断等待仅为 12 个周期。
- 中断和多个异常可由软件触发
- 多个优化用于降低不同异常上下文切换时的中断处理开销。
- 中断/异常屏蔽功能可以将所有的中断和异常（NMI 除外）屏蔽掉，或者将中断/异常屏蔽为某个优先级之下。

为了支持这些特性，NVIC 使用了多个可编程寄存器。这些寄存器经过了存储器映射，而 CMSIS-Core 则为大多数常见的中断控制任务提供了所需的寄存器定义和访问函数（API）。这些访问函数易于使用，而且多数可用于 Cortex-M0 等其他

Cortex-M 处理器。

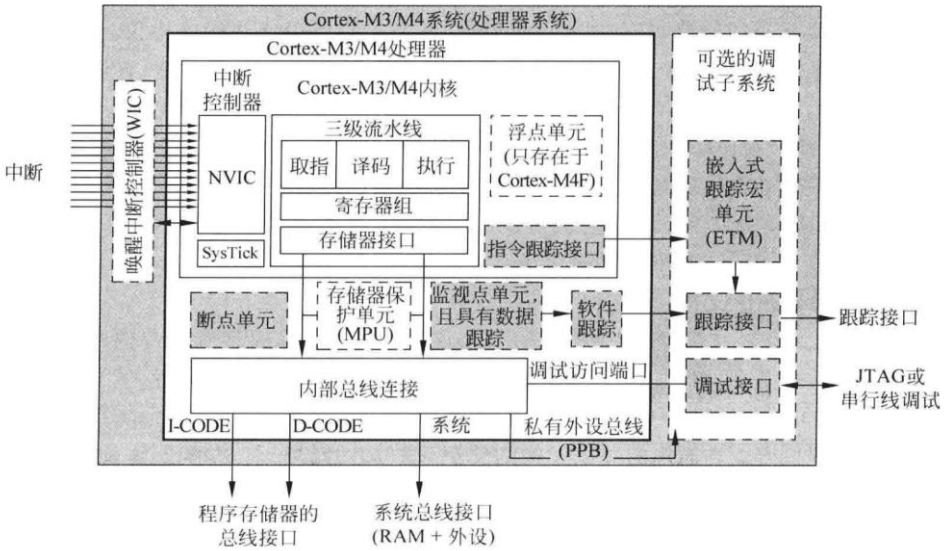


图 1-6　M4 处理器框图

向量表为系统存储器的一部分，其中存有中断和系统异常的起始地址。向量表默认位于存储器空间的开头（地址 0x0），不过，若需要，向量表偏移可以在运行时变为其他值。对于大多数应用程序，向量表可以在编译时被设置为应用程序映像的一部分，且在运行时保持不变。

Cortex-M3 或 Cortex-M4 设备实际支持的中断数量由微控制器供应商在设计芯片时确定。

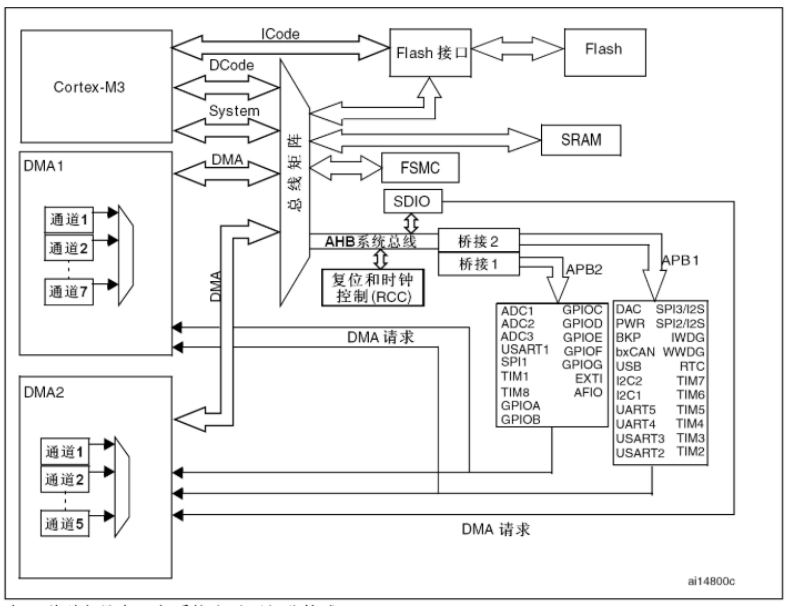


图 1-7 内核与外设连接示意图

跟中断相关的寄存器是 M4 内核中的 xPSR 寄存器，可以通过汇编命令进行访问和修改。

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T		GE*	ICI/IT		异常编号				

*GE在Cortex-M4等ARMv7E-M处理器中存在，在Cortex-M3处理器中则不可用。

位	描　　述
N	负标志
Z	零标志
C	进位(或者非借位)标志
V	溢出标志
Q	饱和标志(ARMv6-M 中不存在)
GE[3:0]	大于或等于标志，对应每个字节通路(只存在于 ARMv7E-M，ARMv6-M 或 Cortex-M3 中则不存在)
ICI/IT	中断继续指令(ICI)位，IF-THEN 指令状态位用于条件执行(ARMv6-M 中则不存在)
T	Thumb 状态，总是 1，清除此位会引起错误异常
异常编号	表示处理器正在处理的异常

图 1-8 xPSR 寄存器描述

此外还有三个特殊寄存器用于异常或中断的屏蔽：PRIMASK、FAULTMASK 和 BASEPRI 寄存器。

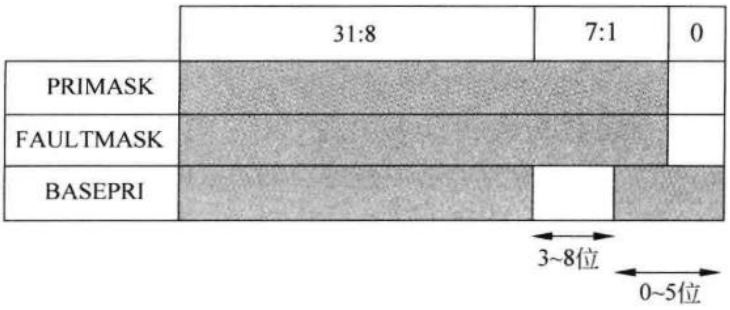


图 1-9 PRIMASK、FAULTMASK 和 BASEPRI 寄存器

每个异常（包括中断）都具有一个优先等级，数值小的优先级高，而数值大的则优先级低。这些特殊寄存器可基于优先等级屏蔽异常，只有在特权访问等级才可以对它们进行操作（非特权状态下的写操作会被忽略，而读出则会返回 0）。它们默认全部为 0，也就是屏蔽（禁止异常/中断）不起作用。

PRIMASK 寄存器为 1 位宽的中断屏蔽寄存器。在置位时，它会阻止不可屏蔽中断（NMI）和 HardFault 异常之外的所有异常（包括中断）。实际上，它是将当前异常优先级提升为 0，这也是可编程异常/中断的最高优先级。

PRIMASK 最常见的用途为，在时间要求很严格的进程中禁止所有中断，在该进程完成后，需要将 PRIMASK 清除以重新使能中断。

FAULTMASK 和 PRIMASK 非常类似，不过它还能屏蔽 HardFault 异常，它实际上是将异常优先级提升到了一 1。错误处理代码可以使用 FAULTMASK 以免在错误处理期间引发其他的错误（只有几种）。例如，FAULTMASK 可用于旁路 MPU 或屏蔽总线错误（这些都是可配置的），这样，错误处理代码执行修复措施也就更加容易了。与 PRIMASK 不同，FAULTMASK 在异常返回时会被自动清除。

为使中断屏蔽更加灵活，ARMv7-M 架构还支持 BASEPRI，该寄存器会根据优先等级屏蔽异常或中断。BASEPRI 的宽度取决于设计实际实现的优先级数量，这是由微控制器供应商决定的。大多数 Cortex-M3 或 Cortex-M4 微控制器都有 8 个（3 位宽）或 16 个可编程的异常优先级，此时，BASEPRI 的宽度就相应地为 3 位或 4 位。BASEPRI 为 0 时就不会起作用，当被设置为非 0 数值时，它就会屏蔽具有相同或更低优先级的异常（包括中断），而更高优先级的则还可以被处理器接受。

ii. 异常和中断的区别

异常：异常是会改变程序流的事件，当其产生时，处理器会暂停当前正在执行的任务，转而执行一段被称作异常处理的程序。异常处理执行完后，处理器会继续正常地执行程序

中断：对于 ARM 架构，中断就是异常的一种，一般由外设或外部输入产生，也可由软件触发

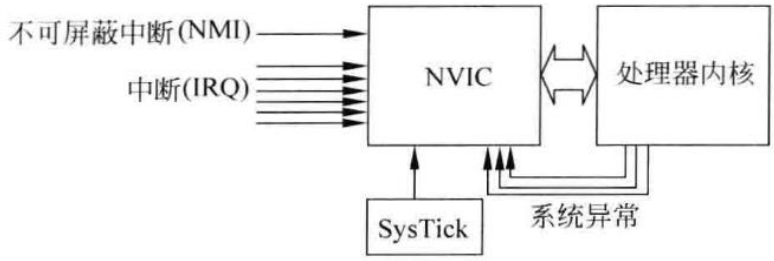


图 1-10 NVIC 与各种中断源

对于其中的异常编号，会在异常发生时被硬件自动写入 IPSR 寄存器内部，异常入口可以从这个编号读取到异常处理的入口地址。

复位也是一种特殊的异常，当 CPU 从复位中推出的时候，会在线程模式下执

行复位处理。

iii. NVIC 相关内容

NVIC 在软件的角度上看，是一个被映射成外设的处理器内部模块，编程人员可以像访问外设一样去操控配置 NVIC 模块。

它的特性有：

1. 灵活的中断和异常管理：除了 NMI 都能禁止，可处理多种中断源
2. 通过可编程优先级，支持中断嵌套（抢占）→咬尾机制
3. 向量化异常处理入口地址
4. 能够通过 PRIMASK 等直接屏蔽很多中断

表 7.9 用于中断控制的 NVIC 寄存器列表

地址	寄存器	CMSIS-Core 符号	功 能
0xE000E100～ 0xE000E11C	中断设置使能寄存器	NVIC-> ISER [0]～ NVIC->ISER [7]	写 1 设置使能
0xE000E180～ 0xE000E19C	中断清除使能寄存器	NVIC-> ICER [0]～ NVIC->ICER [7]	写 1 清除使能
0xE000E200～ 0xE000E21C	中断设置挂起寄存器	NVIC-> ISPR [0]～ NVIC->ISPR [7]	写 1 设置挂起状态

地址	寄存器	CMSIS-Core 符号	功 能
0xE000E280～ 0xE000E29C	中断清除挂起寄存器	NVIC-> ICPR [0]～ NVIC->ICPR [7]	写 1 清除挂起状态
0xE000E300～ 0xE000E31C	中断活跃位寄存器	NVIC->IABR [0]～ NVIC->IABR [7]	活跃状态位，只读
0xE000E400～ 0xE000E4EF	中断优先级寄存器	NVIC->IP [0]～ NVIC->IR [239]	每个中断的中断优先级（8 位宽）
0xE000EF00	软件触发中断寄存器	NVIC->STIR	写中断编号设置相应中断的挂起状态

系统复位后，中断被禁止，所有中断优先级为 0（最高），所有中断挂起状态清零。

NVIC 配置：

1. 设置中断优先级分组：NVIC_SetPriorityGrouping(5)
2. 设置中断优先级（默认为 0 最高），假设位宽为 4：

   NVIC_SetPriority(Timer0_IRQn, 0xC)
3. 使能中断：NVIC_EnableIRQ(Timer0_IRQn)

iv. EXTI 外设

EXIT 外设为 M4 内核外的一个功能模块，负责响应所有外部中断信号。在下图中，中断信号来自于输入线，通过边沿检测电路捕捉到后，如果中断屏蔽寄存器未屏蔽该中断，则信号输入中断挂起请求寄存器，然后传至 NVIC 中进行处理。

图 32. 外部中断/事件控制器框图

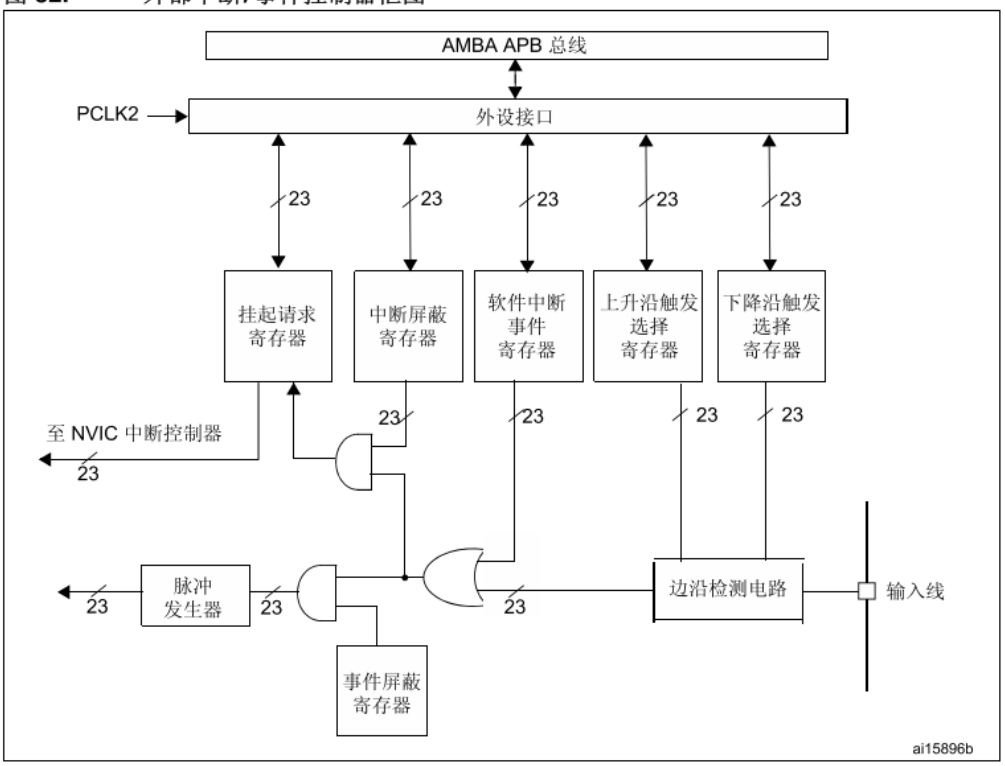


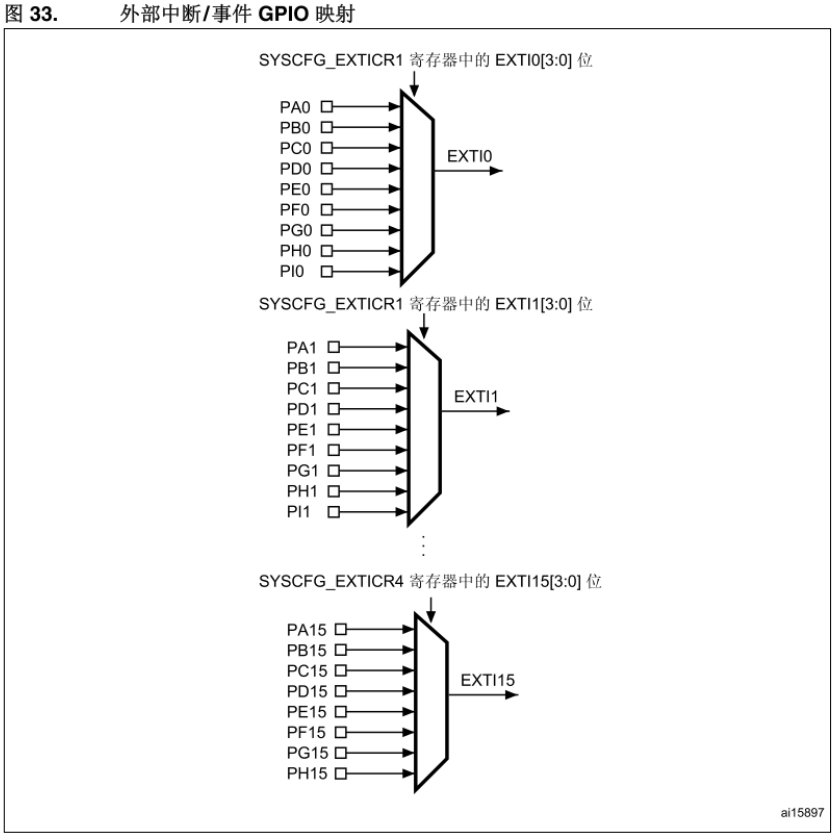
图 1-11 EXTI 触发框图

EXIT 通过外部中断/事件线映射的方式，将多大 140 个 GPIO 口通过一下方式链接到 EXIT 线上，用于收集中断信号。

10.2.5 外部中断/事件线映射

多达 140 个 GPIO（STM32F405xx/07xx 和 STM32F415xx/17xx）通过以下方式连接到 16 个外部中断/事件线：

图 33. 外部中断/事件 GPIO 映射



另外七根 EXTI 线连接方式如下：

- EXTI 线 16 连接到 PVD 输出
- EXTI 线 17 连接到 RTC 闹钟事件
- EXTI 线 18 连接到 USB OTG FS 唤醒事件
- EXTI 线 19 连接到以太网唤醒事件
- EXTI 线 20 连接到 USB OTG HS（在 FS 中配置）唤醒事件
- EXTI 线 21 连接到 RTC 入侵和时间戳事件
- EXTI 线 22 连接到 RTC 唤醒事件

v. 中断流程

中断输入和挂起

挂起：顾名思意，在处理中断 A 的时候，把中断 A 禁止了，然后在退出中断 A 的处理之前，又来了一个 A，则 A 自动被挂起，中断处于挂起状态，“等待处理”、“排队”，不会一直产生请求信号，或者处理中途请求撤销这种情况

中断入口处，多个寄存器会被压栈，同时，ISR 的起始地址会被从向量表中取出。处理完后，执行“异常返回”，之前压栈的寄存器纷纷出栈，被中断程序继续执行。中断活跃状态被清除。

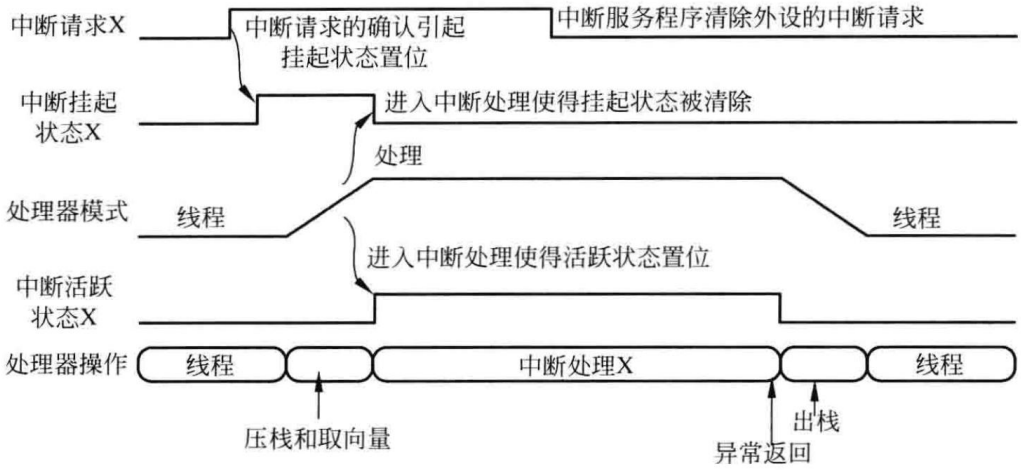


图 1-12 中断挂起和激活行为

进入中断服务程序后，务必要进行中断源的清除，如果不清除，则会产生以下情况。

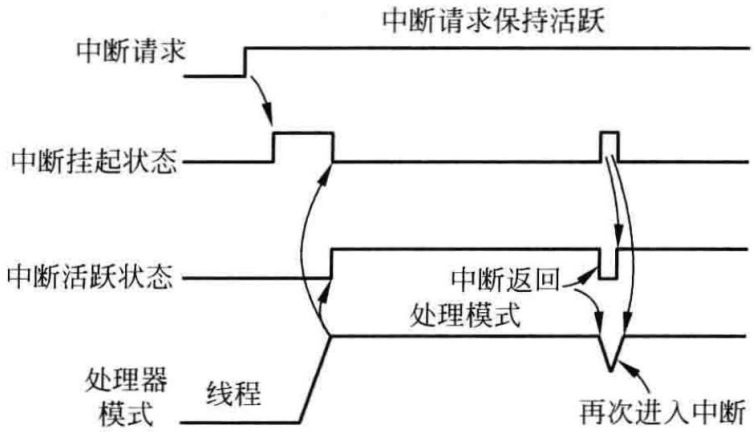


图 1-13 异常退出后再次挂起

接收异常请求：满足下面条件才会接收请求

异常的进入：

1. 硬件自动将多个寄存器和返回地址被压入当前使用的栈！线程模式 PSP 就用 PSP 栈，线程模式非 PSP 就用 MSP；

2. 取异常向量 ISR：拿到异常后取访问向量表，取得起始地址（和压栈并行执

行）；

3. 确定异常处理起始地址（准备给 PC）后执行程序，准备取出，但还没取；

4. 更新 NVIC 寄存器和内核寄存器：xPSR、LR、PC、SP；

注：根据实际使用的栈，MSP/PSP 会自动调整（自动入栈后指针变化），PC 会更新、LR 更新为 EXC_RETURN（异常返回）；

异常处理：

1. 用 MSP，特权等级；

2. 高优先级中断接收后会发生抢占（嵌套）；

3. 相同或低优先级中断接收后挂起；

4. 处理完后将 LR（EXC_RETURN）给 PC，触发异常返回机制；

异常返回：

1. 由 PC 里的 EXC_RETURN 所包含的地址所触发，这是一个流程，这个地址本身找不到（用不执行 XN 存储器属性）；

2. 触发异常返回后，处理器访问栈空间中在异常期间压入栈的寄存器，并将其恢复；

3. 更新 NVIC 寄存器、xPSR、SP、CONTROL 寄存器；

4. 出栈的时候也会取之前程序的指令（流水线操作，加快速度）；

d) OS 时间驱动机制设计

i. 时钟与晶振

晶振是利用石英晶体(二氧化硅的结晶体)的压电效应制成的一种谐振器件，若在石英晶体的两个电极上加一电场，晶片就会产生机械变形。反之，若在晶片的两侧施加机械压力，则在晶片相应的方向上将产生电场，这种物理现象称为压电效应。如果在晶片的两极上加交变电压，晶片就会产生机械振动，同时晶片的机械振动又会产生交变电场。在一般情况下，晶片机械振动的振幅和交变电场的振幅非常微小，但当外加交变电压的频率为某一特定值时，振幅明显加大，比其他频率下的振幅大得多，这种现象称为压电谐振，它与 LC 回路的谐振现象十分相似。它的谐振频率与晶片的切割方式、几何形状、尺寸等有关。

ii. 时钟的分类与选用

众所周知，STM32 可选的时钟有四个，即 HSE(高速外部时钟),HSI(高速内部时钟),LSE(低速外部时钟),LSI(低速内部时钟)。

而如何选用是一个非常重要的问题。时钟系统一般是分为 RTC 时钟和系统时钟的。

1. RTC 时钟： 实时时钟。如果供电，它会按照自己的精确等级运行的，主要用来做日期时间的显示用。

2. 系统时钟：是一个存储于系统内存中的逻辑时钟。用于系统的计算，比如超时产生的中断异常，超时计算就是由系统时钟计算的。这种时钟在系统掉电或重新启动时每次会被清除。

LSE 和 LSI 主要是用于配置 RTC 时钟(独立看门狗)的，HSE 和 HSI 则是用于配置系统时钟的。本次讨论的重点在系统时钟，故仅讨论后者。而 HSE 较 HSI 更稳定，所以一般为首选。

不同的开发板 HSE 时钟源提供不同，对于 Nucleo 开发板，没有焊 X3，它的时钟来自 ST-Link 上的 HSE。

图 1-14 Nucleo 开发板晶振示意

iii. M4 内核时钟树

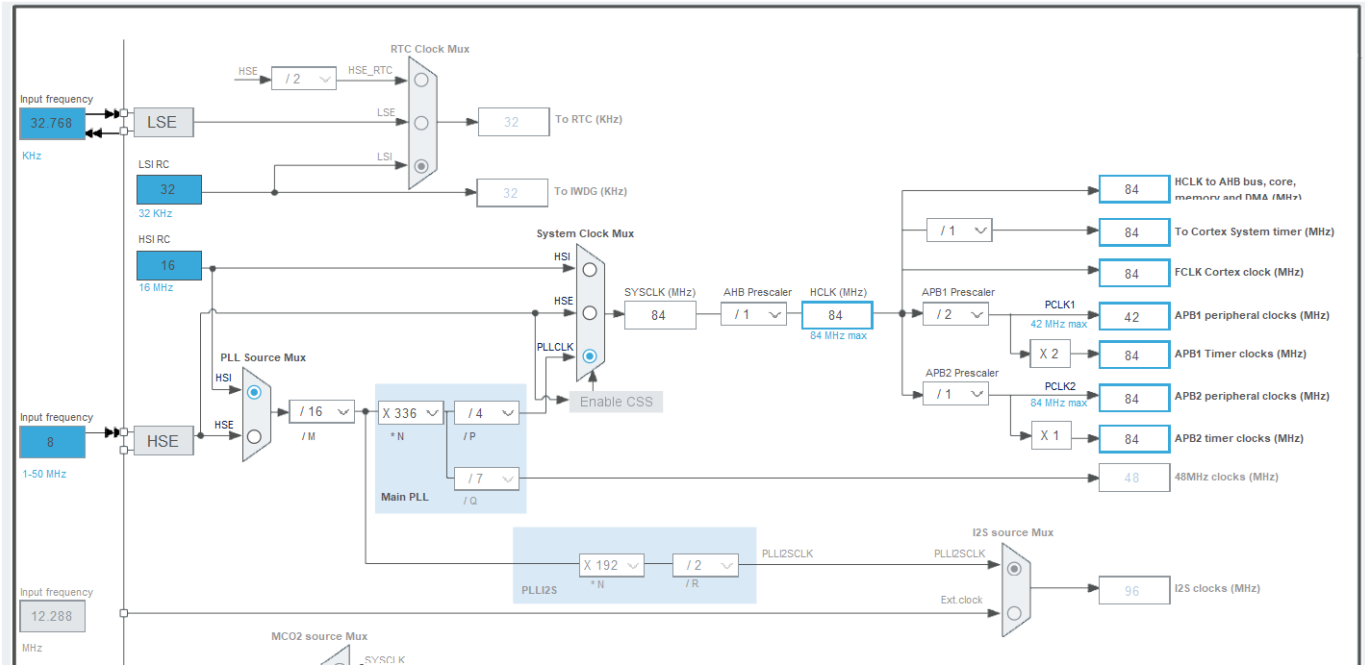


图 1-15 时钟树

这个是通过 CubeMx 打开的图形化时钟配置界面。系统时钟的来源有三：HSI、HSE 和 PLL。

PLL:一般的晶振由于工艺与成本原因，做不到很高的频率，而在需要高频应用时，由相应的器件 VCO(输出频率与输入控制电压有对应关系的振荡电路)，实现转成高频，但并不稳定，故利用锁相环路就可以实现稳定且高频的时钟信号。

iv. 时钟初始化

在标准模板中有关于时钟初始化的代码 SystemInit，它由汇编启动文件启动，在 main 函数之前完成，我们重写时钟初始化函数。

代码 1-1 时钟初始化 1

```
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};//OSC 结构体
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};//CLK 结构体
  __HAL_RCC_PWR_CLK_ENABLE();//启动时钟电源
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);//
```

```
电源模式配置
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;//OSC 配置
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 12;
  RCC_OscInitStruct.PLL.PLLN = 96;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 4;
//计算时钟为 100MHz
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)//执行配置方案
  {
    Error_Handler();
  }
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK

|RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;//CLK 配置
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_3) != HAL_OK)//执
行 CLK 配置
  {
    Error_Handler();
  }
}
```

首先完成关于时钟的初始化。

在这一部分中首先需要打开 HSI。如果单时钟失效了则整个单片机系统停止工作，所以 STM32 是有保险的。当系统选择为外部时钟时，若外部时钟失效，系统会智能地切换到内部时钟，程序正常跑动（虽然很慢，但可以通过 CSS 倍频）。所以开启内部时钟成了初始化的必要工作。

在初始化以后，正式进入时钟配置部分，即对应 system_stm32f4xx.c 里的 SetSysClock()。

这一部分可以大致分为 HSE 和 PLL 的选配。

HSE 的部分如下

先定义了两个变量，用来辅助判断。StartUPCounter 放在[SP,#0X04]中，HSEStatus 放在[SP,#0X00]中。

时钟的每一步都是严谨的，我们必须等到其确认已经稳定后才可以进行下一步。

代码 1-2 时钟初始化 2

```
//开始配置 HSE
        __HAL_RCC_HSE_CONFIG(RCC_OscInitStruct->HSEState);
      //检测 HSE 状态
       if((RCC_OscInitStruct->HSEState) != RCC_HSE_OFF)
       {
        //获取 tick
         tickstart = HAL_GetTick();
         //等待 HSE 就绪
         while(__HAL_RCC_GET_FLAG(RCC_FLAG_HSERDY) == RESET)
         {
           if((HAL_GetTick() - tickstart ) > HSE_TIMEOUT_VALUE)
           {
             return HAL_TIMEOUT;
           }}}
```

然后是关于 PLL 的配置

启动 PLL 的设置一次是在 RCC->CR 处置位 PLLON，使能。一次是在 RCC->CFGR 处置位 SW，选择 PLL 为时钟源。

代码 1-3 时钟初始化 3

```
  //启动主 PLL
        __HAL_RCC_PLL_ENABLE();
        //获取 tick
        tickstart = HAL_GetTick();
        //等待 PLL 就绪
        while(__HAL_RCC_GET_FLAG(RCC_FLAG_PLLRDY) == RESET)
        {
          if((HAL_GetTick() - tickstart ) > PLL_TIMEOUT_VALUE)
          {
            return HAL_TIMEOUT;
          }
        }
```

e) OS 启动代码设计

i. 启动概述

uC/OS-II 运行前要把时钟中断的频率设定好，完成各个硬件模块的 HAL 初始化，接着就需要执行 OSInit 做一些准备工作, 比如初始化 TCB 列表，就绪对列表和事件控制表，然后我们就要创建四轴飞行器各个模块的任务，接着调用 OSStart 开始整个系统运行，在 OSStart 最开始会先调用 OSSchedNew 计算出优先级最高的任务, 然后执行 OSStartHighRdy 函数，OSStartHighRdy 触发一个 PendSV 软件中断，在 OSPendSV 中将优先级最高的任务调度给 CPU 执行，随后整个程序就能顺利运行起来了。

ii. 栈初始化与 SP 指针

uC/OS-II 系统设计有多个堆栈，包括一个系统堆栈和多个线程堆栈，其中系统堆栈是 OS 运行时使用的，线程堆栈是每个任务存放数据用的。同时 Cortex-M4 内核寄存器组也设置了 2 个堆栈指针，MSP 和 PSP，其中 MSP 是主堆栈指针，PSP 是线程堆栈指针，和 uC/OS-II 系统对应。执行不同的命令可以交替使用这两个堆栈指针，所以我们就将 MSP 作为 OS 运行的系统堆栈指针，PSP 作为正在运行的任务的堆栈指针。启动后 CM4 内核芯片默认在 MSP 状态，所以我们只要初始化好 SP 为一块空内存区域的首地址即可作为 uC/OS-II 启动的系统堆栈。

代码 1-4 栈设置代码

```
Stack_Size      EQU     0x400
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp
```

iii. 堆初始化

堆初始化与 stm32 官方提供的 startup 中的堆初始化保持一致，但大小可以增加一倍，防止溢出进入 HardFault。

代码 1-5 设置堆代码

```
Heap_Size       EQU     0x200
                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

iv. 中断向量表

uC/OS-II 需要能够使用 MCU 控制板的时钟中断和 PendSV 中断。所以要根据 OS 的需要修改中断向量表，替换 PendSV 中断和 Systick 中断为 uC/OS-II 自己的函数。

代码 1-6 中断向量表

```
__Vectors       DCD     __initial_sp                ; Top of Stack
                DCD     Reset_Handler               ; Reset Handler
                DCD     NMI_Handler                 ; NMI Handler
                DCD     HardFault_Handler           ; Hard Fault Handler
                DCD     MemManage_Handler           ; MPU Fault Handler
                DCD     BusFault_Handler            ; Bus Fault Handler
                DCD     UsageFault_Handler          ; Usage Fault Handler
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     SVC_Handler                 ; SVCall Handler
                DCD     DebugMon_Handler            ; Debug Monitor Handler
                DCD     0                           ; Reserved
                DCD     PendSV_Handler              ; PendSV Handler
                DCD     SysTick_Handler             ; SysTick Handler

                ; External Interrupts
                DCD     WWDG_IRQHandler             ; Window WatchDog
                DCD     PVD_IRQHandler              ; PVD through EXTI
                ……
                ……
                DCD     SPI5_IRQHandler             ; SPI5

__Vectors_End
```

v. 复位中断

复位中断是优先级最高，飞控板上电就会执行的代码，在这部分我们完成芯片启动，初始化 C 语言执行环境和一部分 uC/OS-II 初始化的必要操作。

代码 1-7 复位中断

```
; Reset handler
Reset_Handler       PROC
                    EXPORT   Reset_Handler              [WEAK]
                    IMPORT   SystemInit
                    IMPORT   __main
                    LDR      R0, =SystemInit
                    BLX      R0
                    LDR      R0, =__main
                    BX       R0
                    ENDP
```

vi. 自定义中断

启动部分我们需要自定义的中断即中断向量表修改的时钟中断和 PendSV 中断。

代码 1-8 PendSV 中断服务程序

```
PendSV_Handler
    CPSID    I                      ; Prevent interruption during context switch
    MRS      R0, PSP                ; PSP is process stack pointer  如果在用 PSP 堆栈,则可以忽略保存寄存器,参考 CM3 权威中的双堆栈
    CBZ      R0, PendSV_Handler_Nosave    ; Skip register save the first time

    ;Is the task using the FPU context? If so, push high vfp registers.
    ;TST      R14, #0x10
    ;IT       EQ
    ;VSTMDBEQ R0!, {S16-S31}

    SUBS     R0, R0, #0x20          ; Save remaining regs r4-11 on process stack
    STM      R0, {R4-R11}
    LDR      R1, =OSTCBCur      ; OSTCBCur->OSTCBStkPtr = SP;
    LDR      R1, [R1]
    STR      R0, [R1]               ; R0 is SP of process being switched out
```

vii. SysTick 中断

SysTick 中断主要用于在操作系统中进行任务的同步，由专门的时钟触发 SysTick，产生 Tick 信号，以此驱动操作系统正常工作下去。

代码 1-9 SysTick 中断服务程序

```
void SysTick_Handler(void)
{
  /* USER CODE BEGIN SysTick_IRQn 0 */
    HAL_IncTick();
    if(OSRunning==1)              //OS 开始跑了,才执行正常的调度处理
    {
        OSIntEnter();             //进入中断
        OSTimeTick();             //调用 ucos 的时钟服务程序
        OSIntExit();                    //触发任务切换软中断
    }
```

viii. 用户堆栈初始化

用户堆栈初始化是首先由用户定义一个类型为 OS_STK，大小足够的数组，然后在创建任务中 OSTaskCreate 调用 OSTaskStkInit 把该堆栈初始化给相应的用户任务。

代码 1-10 用户堆栈初始化与任务创建

```
//设置任务堆栈大小
#define DATA_TRANSFER_STK_SIZE                  128
//任务堆栈
OS_STK DATA_TRANSFER_TASK_STK[DATA_TRANSFER_STK_SIZE];


OSTaskCreate(data_transfer_task,(void*)0,(OS_STK*)&DATA_TRANSFER_TASK_STK[DAT
A_TRANSFER_STK_SIZE-1],DATA_TRANSFER_TASK_PRIO );
```

f) 系统集成与业务层移植

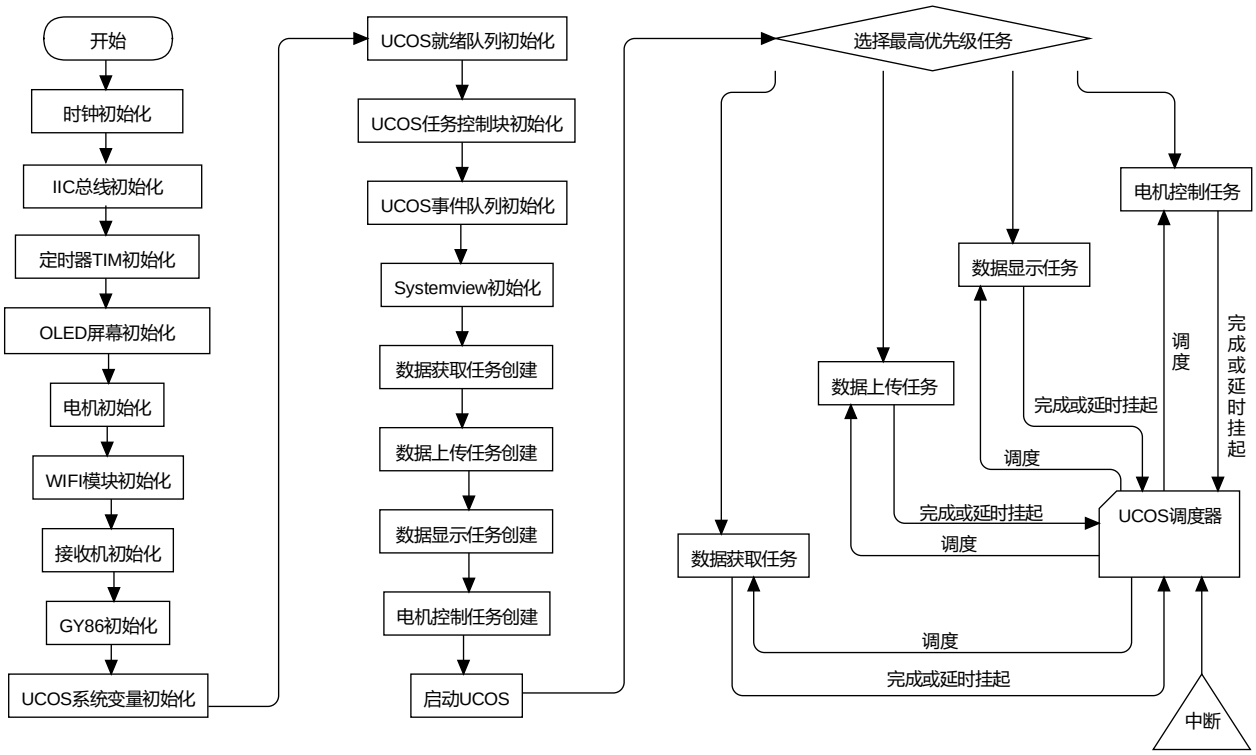


图 1-16 业务层逻辑流程示意图

i. 任务优先级设计

本小组的设置了四个任务，分别为数据获取和融合，电机控制，数据输出和调试，数据向上位机发送四个任务。

目前设计这四个任务的原因主要是飞行器尚不具备稳定飞行的能力，所以目前暂不需要设置 PID 姿态控制相关的任务，但预留了相关任务的添加接口。

这四个任务的优先级设置为数据获取和融合 >电机控制> 数据向上位机发送 > 数据本地输出调试 ，至于为什么如此安排，因为本小组经过讨论认为获取最新数据处理结果是最重要的，然后是对电机的实时控制。其次，数据上传的快慢影响着电脑调试的姿态显示与真实的姿态的延时，需要让这个延时尽可能短，所以将数据上传放在第三位，至于数据的本地显示则不能么重要，只是一个副参考，所以将优先级设置的最低。

ii. 任务创建与初始化

数据获取和融合，电机控制，数据输出和调试，数据向上位机都通过OSTaskCreate 创建，四个任务的堆栈大小均为 128，唯一不同的是优先级，分别为6，8，9，10。

iii. 数据获取任务

数据获取部分主要是从地磁计，陀螺仪和加速度计获取标准平面，角加速度和线加速度，然后对角加速度积分得到实时角度，将线加速度积分得到速度。

采用了互补滤波算法，用陀螺仪来修正静态加速度计的抖动偏差，用加速度计来修正动态陀螺仪的角度偏差，用地磁仪来校准偏航角度。

计算过程中采用了四元素来完成中间过程计算，再将四元数转换为欧拉角，得到最终的解算结果。

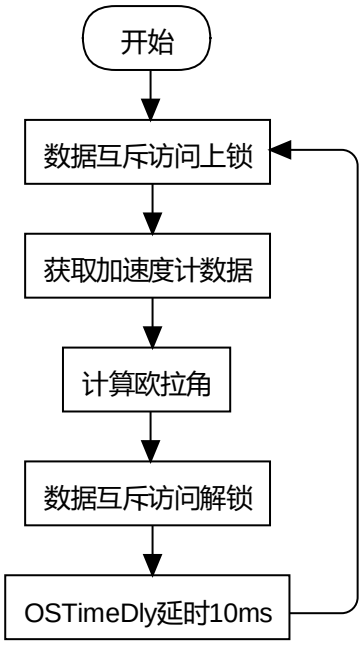


图 1-17 数据获取任务流程图

iv. 电机控制任务

通过中断获取到遥控器控制信息后直接将占空比作用于电机控制，中间暂时省略了 PID 反馈控制环节，作为电机控制的实验参考。

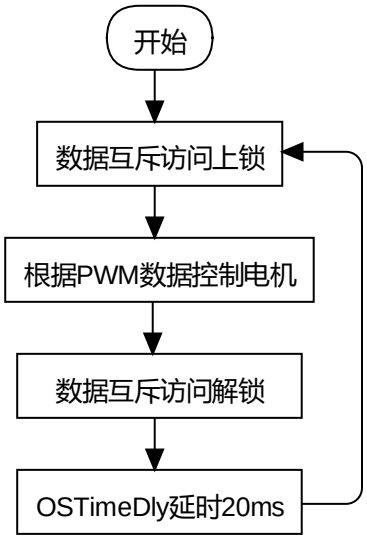


图 1-18 电机控制任务流程图

v. 数据上传任务

数据上传任务主要是将传感器原始数据和解算后的欧拉角数据，以及遥控器各通道数据上传给 PC 的匿名上位机软件，并通过匿名上位机软件对飞行器的飞行姿态进行实时渲染，同时展示飞行器各项数据指标。

匿名上位机是航模爱好者开发的一款开源能显示航模姿态，提供调试辅助的软件。我们需要将相关数据按照相应的数据交互协议构建 Payload，通过串口 WIFI 上传给匿名上位机进行数据解析。

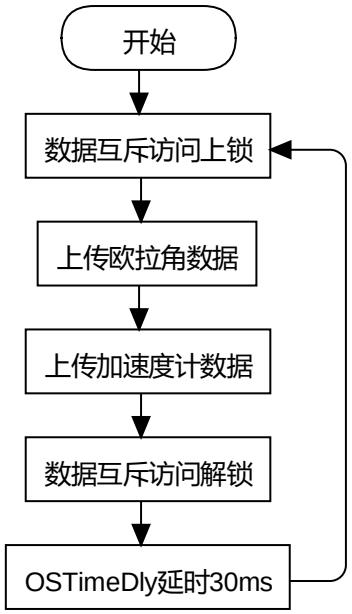


图 1-19 数据上传任务流程图

vi. 数据显示任务

数据显示任务主要是在本地的 OLED 显示屏上显示相关的飞行姿态数据，比如角加速度，线加速度，欧拉角以及 MCU 的温度。

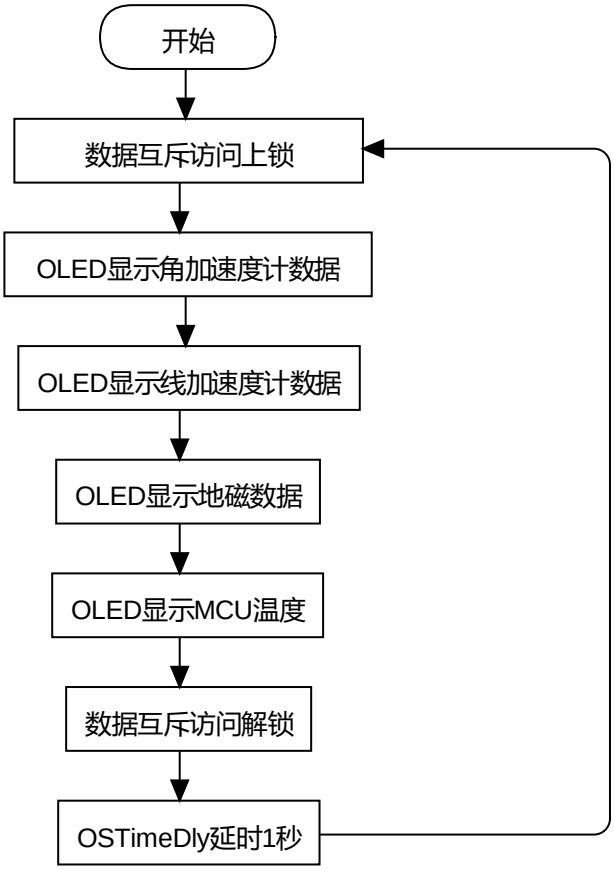
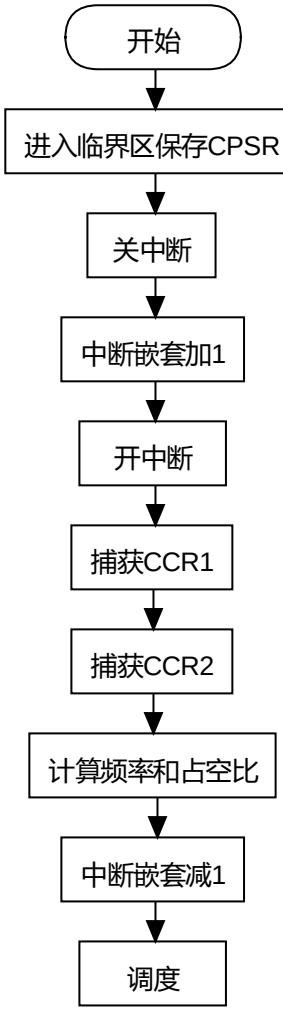


图 1-10 数据显示任务流程图

vii. 中断服务程序

中断服务程序包括第一个学期完成的遥控器接收机数据 TIM PWM 频率和占空比捕获和串口接收中断。



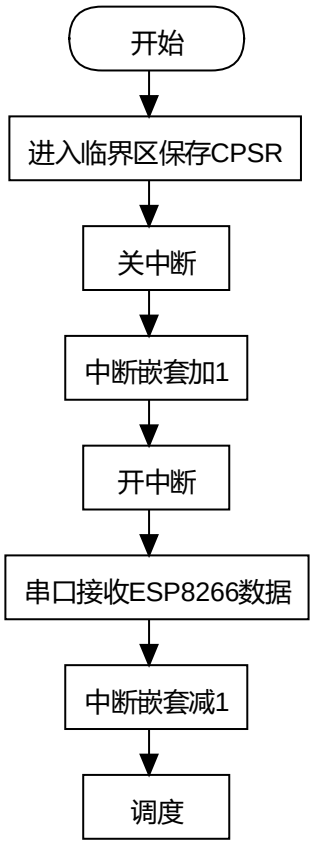


图 1-19,20 接收机信号中断捕获流程图（左）
串口数据传输中断流程图（右）

1.2 针对复杂工程问题的方案实现

一、 任务创建

上位机数据传输任务

代码 2-1 上位机传输任务

```
//DATA_TRANSFER 任务
//设置任务优先级
#define DATA_TRANSFER_TASK_PRIO          9
//设置任务堆栈大小
#define DATA_TRANSFER_STK_SIZE           128
```

```
//任务堆栈
OS_STK DATA_TRANSFER_TASK_STK[DATA_TRANSFER_STK_SIZE];
//任务函数
void data_transfer_task(void *pdata);
```

代码 2-2 板载数据调试显示任务

```
//ON BOARE DEBUG 任务
//设置任务优先级
#define ON_BORAD_DEBUG_TASK_PRIO        10
//设置任务堆栈大小
#define ON_BORAD_DEBUG_STK_SIZE         128
//任务堆栈
OS_STK ON_BORAD_DEBUG_TASK_STK[ON_BORAD_DEBUG_STK_SIZE];
//任务函数
void on_board_debug_task(void *pdata);
```

代码 2-3 数据融合姿态解算任务

```
//DATA FUSION 任务
//设置任务优先级
#define DATA_FUSION_TASK_PRIO       6
//设置任务堆栈大小
#define DATA_FUSION_STK_SIZE        128
//任务堆栈
OS_STK DATA_FUSION_TASK_STK[DATA_FUSION_STK_SIZE];
//任务函数
void data_fusion_task(void *pdata);
```

代码 2-4 电机控制任务

```
//MOTOR CONTROL 任务
//设置任务优先级
#define MOTOR_CONTROL_TASK_PRIO             8
//设置任务堆栈大小
#define MOTOR_CONTROL_STK_SIZE              128
//任务堆栈
OS_STK MOTOR_CONTROL_TASK_STK[MOTOR_CONTROL_STK_SIZE];
//任务函数
void motor_control_task(void *pdata);
```

代码 2-5 任务创建

```
//创建上位机数据传输任务
OSTaskCreate(data_transfer_task,(void*)data_transfer_name,(OS_STK
*)&DATA_TRANSFER_TASK_STK[DATA_TRANSFER_STK_SIZE-
1],DATA_TRANSFER_TASK_PRIO );
```

```
//创建板载数据调试任务
OSTaskCreate(on_board_debug_task,(void *)on_board_debug_name,(OS_STK *)&ON_BORAD_DEBUG_TASK_STK[ON_BORAD_DEBUG_STK_SIZE-1],ON_BORAD_DEBUG_TASK_PRIO );
//创建数据融合姿态解算任务
OSTaskCreate(data_fusion_task,(void *)data_fusion_name,(OS_STK *)&DATA_FUSION_TASK_STK[DATA_FUSION_STK_SIZE-1],DATA_FUSION_TASK_PRIO );
//创建电机控制任务
OSTaskCreate(motor_control_task,(void *)motor_control_name,(OS_STK *)&MOTOR_CONTROL_TASK_STK[MOTOR_CONTROL_STK_SIZE-1],MOTOR_CONTROL_TASK_PRIO );
```

二、 数据融合与姿态解算任务

代码 2-6 Date_Fusion

```
//data_fusion
void data_fusion_task(void *pdata)
{
    while(1)
    {
    OSSemPend(MPU6050_rawData_Mutex,0,&err);
    read_Gyroscope_DPS();
    read_Accelerometer_MPS();
    OSSemPost(MPU6050_rawData_Mutex);
    OSSemPend(Data_Fusion_Result_Mutex,0,&err);
    mpu_dmp_get_data(&pitch,&roll,&yaw);
    OSSemPost(Data_Fusion_Result_Mutex);
    OSTimeDly(10);
    }
}
```

三、 上位机数据传输任务

代码 2-7 Data_Transfer

```
//data_transfer
void data_transfer_task(void *pdata)
{
    while(1)
```

```
    {
    OSSemPend(Receiver_Data_Mutex,0,&err);
    for(int i = 0; i < 6; i++)
    {
        Cap[i] = (Duty[i] * 100 - 5) / 0.05;
    }
    OSSemPost(Receiver_Data_Mutex);
    OSSemPend(Data_Fusion_Result_Mutex,0,&err);
    ANO_DT_Send_Status(roll,pitch,yaw,2333,122,(Cap[5]>50));
    OSSemPost(Data_Fusion_Result_Mutex);
    OSSemPend(MPU6050_rawData_Mutex,0,&err);
    ANO_DT_Send_Senser(Ax,Ay,Az,Gx,Gy,Gz,2333,2333,2333);
    OSSemPost(MPU6050_rawData_Mutex);
    ANO_DT_Send_RCData(0,0,0,0,Cap[0],Cap[1],Cap[2],Cap[3],Cap[4],Cap[5]);
    OSTimeDly(30);
    }
}
```

四、 板载数据显示任务

代码 2-8 On_Board_Debug

```
//on_board_debug
void on_board_debug_task(void *pdata)
{
    while(1)
    {
        OSSemPend(MPU6050_rawData_Mutex,0,&err);
        OLED_Show_3num(Gx, Gy, Gz, 0);
        OLED_Show_3num(Ax, Ay, Az, 1);
        OLED_Show_3num(Mag_x, Mag_y, Mag_z, 2);
        OLED_ShowNum(24, 7, MPU_Get_Temperature(),2,12);
        OSSemPost(MPU6050_rawData_Mutex);
        OSSemPend(Receiver_Data_Mutex,0,&err);
        OLED_Show_3num(Cap[0], Cap[1], Cap[2], 3);
        OLED_Show_3num(Cap[3], Cap[4], Cap[5], 4);
        OSSemPost(Receiver_Data_Mutex);

        OSTimeDly(1000);
    }
}
```

五、 中断服务程序

代码 2-9 接收机汇中断捕获

```
void TIM1_UP_TIM10_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&htim1);
}
void TIM1_TRG_COM_TIM11_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim1);
  HAL_TIM_IRQHandler(&htim11);
}
void TIM2_IRQHandler(void)
{
    OS_CPU_SR    cpu_sr = 0u;
    OS_ENTER_CRITICAL();
    OSIntEnter();
    OS_EXIT_CRITICAL();
    HAL_TIM_IRQHandler(&htim2);
    OSIntExit();
}
void TIM4_IRQHandler(void)
{
    OS_CPU_SR    cpu_sr = 0u;
    OS_ENTER_CRITICAL();
    OSIntEnter();
    OS_EXIT_CRITICAL();
    HAL_TIM_IRQHandler(&htim4);
    OSIntExit();
}
```

代码 2-19 串口接收中断

```
void USART1_IRQHandler(void)
{
    OS_CPU_SR    cpu_sr = 0u;
    OS_ENTER_CRITICAL();
    OSIntEnter();
    OS_EXIT_CRITICAL();
    HAL_UART_IRQHandler(&huart1);
    OSIntExit();
}
```

第二章 系统测试

2.1 SystemView

SystemView 是一个用于分析嵌入式系统的工具。SystemView 可以完整的深入观察一个应用程序的运行时行为。

由两部分组成：

- SystemView 的 PC 端程序：收集目标板上传的信息
- SystemView 嵌入式端程序：分析嵌入式系统的行为。

类似 BLE，首先上位机 PC 端有一个处理程序，下位机 MCU 上需增加部分代码，用于记录嵌入式系统的一些数据，通过连接的仿真器接口将数据传输到上位机，然后 PC 端的处理程序处理这些数据，由于数据传输使用的是 SEGGER J-link 的实时传输技术（RTT），所以 SystemView 可以实时分析和展示数据。

分析的内容包括中断、任务、软件定时器执行的时间，切换的时间，以及发生的事件等，这些分析都是实时的，丝毫不影响下位机的运行。这样就可以验证我们设计的整个嵌入式系统是否按照我们的预期在工作，比如任务的切换逻辑，中断的触发等。

具体步骤如下。

一、 将 SystemView 源码添加到工程

我们在工程目录下建立三个新的文件夹，分别为 Config，SEGGER 和 OS。将源码包中这些文件添加到工程下文件夹中。

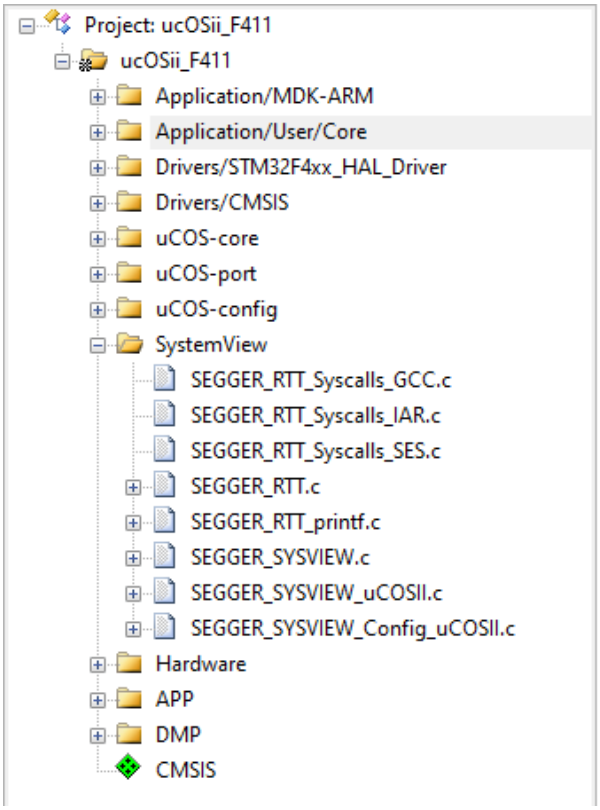


图 2-1 SystemView 添加结构

二、 ST 开发板相关配置

用以下代码覆盖 SEGGER_SYSVIEW_Conf.h

代码 2-1 SEGGER_SYSVIEW_Conf.h

```
#ifndef SEGGER_SYSVIEW_CONF_H #define SEGGER_SYSVIEW_CONF_H
// 事件时间戳
#define SEGGER_SYSVIEW_GET_TIMESTAMP() ((*(U32 *)(0xE0001004)) >> 4)
#define SEGGER_SYSVIEW_TIMESTAMP_BITS 28
// 中断 ID
#define SEGGER_SYSVIEW_ID_BASE 0x20000000
#define SEGGER_SYSVIEW_ID_SHIFT 0
#define SEGGER_SYSVIEW_GET_INTERRUPT_ID() ((*(U32 *)(0xE000ED04)) & 0x1FF)
#define SEGGER_SYSVIEW_UNLOCK() SEGGER_RTT_UNLOCK()
#endif
```

三、 uC/OS II 相关配置

开启记录选项。

代码 2-2 os_cfg.h

```
#define OS_TRACE_EN 1
#define OS_TRACE_API_ENTER_EN 1
#define OS_TRACE_API_EXIT_EN 1
#define OS_TIME_DLY_HMSM_EN 1u /* Include code for OSTimeDlyHMSM() */
#define OS_TIME_DLY_RESUME_EN 1u /* Include code for OSTimeDlyResume() */
#define OS_TIME_GET_SET_EN 1u /* Include code for OSTimeGet() and OSTimeSet() */
#define OS_TIME_TICK_HOOK_EN 1u /* Include code for OSTimeTickHook() */
```

代码 2-3 os_cfg_trace.h

```
#define OS_CFG_TRACE_MAX_TASK 32u
#define OS_CFG_TRACE_MAX_RESOURCES 128u
```

代码 2-4 SEGGER_RTT_Conf.h

```
#ifndef BUFFER_SIZE_UP #define BUFFER_SIZE_UP (4096)
#endif
```

代码 2-5 SEGGER_SYSVIEW_Conf.h

```
#define SEGGER_SYSVIEW_RTT_BUFFER_SIZE 4096u
```

代码 2-6 SEGGER_SYSVIEW_Config_uCOSII.c

```
#define SYSVIEW_APP_NAME "ARM Cortex-M Demo"
#define SYSVIEW_DEVICE_NAME "Cortex-M Device"

// Frequency of the timestamp. Must match SEGGER_SYSVIEW_GET_TIMESTAMP in SEGGER_SYSVIEW_Conf.h
#define SYSVIEW_TIMESTAMP_FREQ ((*(U32 *)(0xE0001004)) >> 4)

// System Frequency. SystemcoreClock is used in most CMSIS compatible projects.
#define SYSVIEW_CPU_FREQ ((*(U32 *)(0xE0001004)) >> 4)

// The lowest RAM address used for IDs (pointers)
#define SYSVIEW_RAM_BASE (0x20000000)
```

四、 集成 SEGGER SystemView

代码 2-7 main.c

```
#include "SEGGER_SYSVIEW.h"
#include "os_trace_events.h"
//在 OS 初始化后添加
OS_TRACE_INIT();
OS_TRACE_START();
```

添加信息记录探针：

代码 2-8 OSTimeDly()

```
if(OSTCBCur->OSTCBPrio!=OS_TASK_IDLE_PRIO)
{
    SEGGER_SYSVIEW_OnTaskStopReady (OSTCBCur, ticks);
}
```

代码 2-9 OS_Sched()

```
if(OSPrioHighRdy==OS_TASK_IDLE_PRIO)
{
    SEGGER_SYSVIEW_OnIdle();
}else
{
    SEGGER_SYSVIEW_OnTaskStartExec(OSTCBHighRdy);
}
if(OSPrioHighRdy==OS_TASK_IDLE_PRIO)
{
    SEGGER_SYSVIEW_OnIdle();
}else
{
    SEGGER_SYSVIEW_OnTaskStartExec(OSTCBCur);
}
```

代码 2-10 OSTaskCreate()

```
SEGGER_SYSVIEW_TASKINFO Info;
memset(&Info, 0, sizeof(Info));
Info.Prio = prio;
Info.StackBase = (unsigned long)ptos;
Info.sName = (const char*)p_arg;
```

代码 2-11 ISR

```
SEGGER_SYSVIEW_RecordEnterISR ();
```

```
ISR;
SEGGER_SYSVIEW_RecordExitISR ();
```

2.2 操作系统运行情况测试与分析

系统信息：

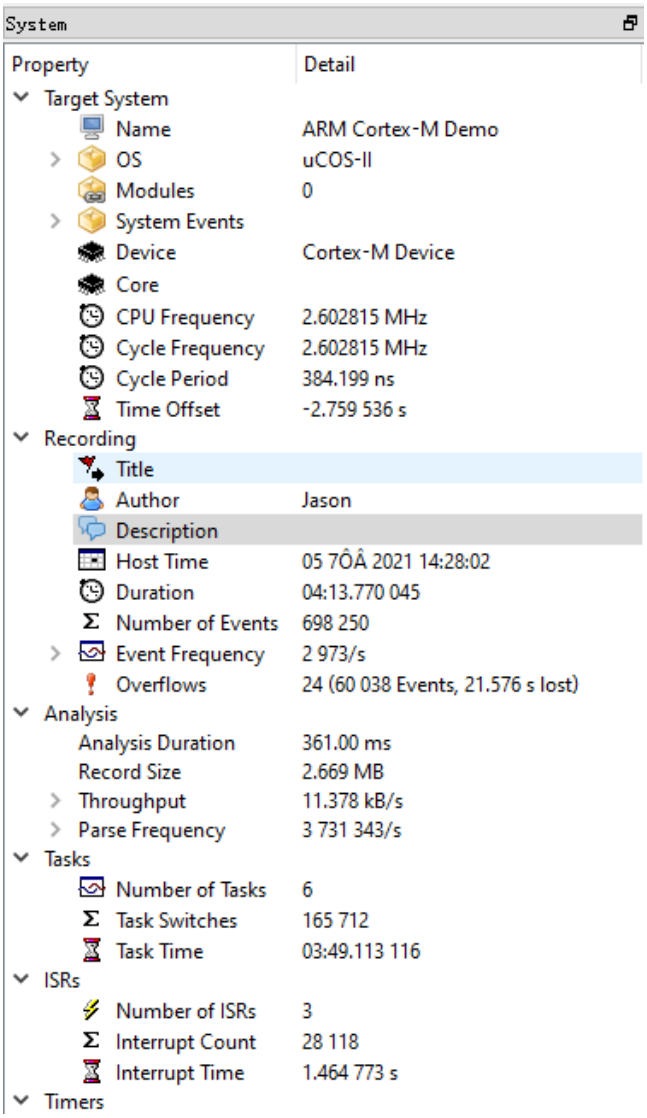


图 2-1 系统信息

从中我们可以获取到整个系统的基础信息，包括了硬件部分的 MCU 架构、时钟频率，同时也可以获取软件部分的任务数量、切换次数、执行时间以及中断数量、触发的次数和执行时间。

任务详情：

这里以单个任务为单位，展现出了每个任务在操作系统调度下的运行情况，

主要体现在了执行次数、执行时间以及 CPU 占用率上。

Contexts

Name	Type	Stack Information	Activations	Total Blocked Time	Total Run Time	Time Interrupted	CPU Load	Last Run Time	Min Run Time	M
SysTick	#15		0		0.000 000 000 s	0.000 000 ms	0.00 %	0.000 000 ms	0.000 000 ms	
ISR 46	#46		14 482		0.282 786 137 s	0.000 000 ms	0.11 %	0.029 583 ms	0.010 373 ms	
ISR 44	#44		13 636		1.184 138 711 s	2.151 517 ms	0.47 %	0.017 673 ms	0.009 221 ms	
Scheduler			11		0.000 109 881 s	0.000 000 ms	0.00 %	0.009 221 ms	0.009 221 ms	
Task 0x12F4	@0	0 @ 0x00000000	96 713	0.000 000 000 s	200.967 830 983 s	1 290.686 430 ms	78.68 %	2.365 516 ms	0.000 384 ms	
Task 0x12A0	@0	0 @ 0x00000000	439	0.000 000 000 s	0.751 303 493 s	3.399 012 ms	0.29 %	0.823 724 ms	0.028 047 ms	12
Data Fusion	@6	0 @ 0x200007DC	12 629	0.000 000 000 s	7.816 084 124 s	55.734 272 ms	3.06 %	0.934 757 ms	0.000 384 ms	
Motor Control	@8	0 @ 0x200009DC	6 448	0.000 000 000 s	13.544 730 609 s	51.768 182 ms	5.32 %	2.311 344 ms	0.010 373 ms	
Data Transfer	@9	0 @ 0x200003DC	25 801	0.000 000 000 s	6.150 342 994 s	42.580 437 ms	2.41 %	2.307 886 ms	0.000 384 ms	
On Board Debug	@10	0 @ 0x200005DC	23 682	0.000 000 000 s	1.335 608 946 s	8.616 056 ms	0.52 %	0.816 040 ms	0.000 384 ms	
Idle			48 292		1.752 668 169 s	11.891 356 ms	0.69 %	0.010 373 ms	0.007 684 ms	

图 2-2 任务详情

任务执行 Log：

此处记录了每一个任务发生的时间，为我们的分析提供了一个详细的时间表，可以准确地了解操作系统在什么时候做了什么工作。

#	Time	Context	Event	Detail
179656	1:19.308 205 155	Task 0x12F4	Task Run	Runs for 2.364 ms
179657	1:19.310 540 319	Data Transfer	Task Run	Runs for 36.499 us
179658	1:19.310 548 387	Data Transfer	Task Run	Runs for 36.499 us
179659	1:19.310 576 818	Task 0x12F4	Task Run	Runs for 1.264 ms
179660	1:19.310 584 502	Task 0x12F4	Task Run	Runs for 1.264 ms
179661	1:19.310 599 486	Task 0x12F4	Task Run	Runs for 1.264 ms
179662	1:19.310 606 401	Task 0x12F4	Task Run	Runs for 1.264 ms
179663	1:19.311 830 460	Task 0x12F4	Task Block	Data Transfer, Reason = 20
179664	1:19.311 840 834	Data Transfer	Task Run	Runs for 1.101 ms
179665	1:19.311 847 749	Data Transfer	Task Run	Runs for 1.101 ms
179666	1:19.312 175 087	ISR 46	ISR Enter	Runs for 18.826 us
179667	1:19.312 193 913	ISR 46	ISR Exit	Returns to Data Transfer
179668	1:19.312 205 055	ISR 46	ISR Enter	Runs for 10.758 us
179669	1:19.312 215 812	ISR 46	ISR Exit	Returns to Data Transfer
179670	1:19.312 226 570	ISR 44	ISR Enter	Runs for 40.341 us
179671	1:19.312 266 911	ISR 44	ISR Exit	Returns to Data Transfer
179672	1:19.312 941 949	On Board Debug	Task Run	Runs for 36.499 us
179673	1:19.312 949 633	On Board Debug	Task Run	Runs for 36.499 us
179674	1:19.312 978 448	Task 0x12F4	Task Run	Runs for 2.364 ms
179675	1:19.312 986 132	Task 0x12F4	Task Run	Runs for 2.364 ms
179676	1:19.313 001 116	Task 0x12F4	Task Run	Runs for 2.364 ms
179677	1:19.313 008 032	Task 0x12F4	Task Run	Runs for 2.364 ms
179678	1:19.315 343 196	On Board Debug	Task Run	Runs for 36.499 us
179679	1:19.315 350 880	On Board Debug	Task Run	Runs for 36.499 us
179680	1:19.315 379 695	Task 0x12F4	Task Run	Runs for 2.364 ms
179681	1:19.315 387 379	Task 0x12F4	Task Run	Runs for 2.364 ms
179682	1:19.315 402 362	Task 0x12F4	Task Run	Runs for 2.364 ms
179683	1:19.315 409 278	Task 0x12F4	Task Run	Runs for 2.364 ms
179684	1:19.315 776 957	ISR 44	ISR Enter	Runs for 232.441 us
179685	1:19.316 009 398	ISR 44	ISR Exit	Returns to Task 0x12F4
179686	1:19.317 004 090	ISR 46	ISR Enter	Runs for 29.583 us
179687	1:19.317 033 673	ISR 46	ISR Exit	Returns to Task 0x12F4
179688	1:19.317 744 442	On Board Debug	Task Run	Runs for 37.652 us
179689	1:19.317 752 126	On Board Debug	Task Run	Runs for 37.652 us

图 2-3 任务执行 log

任务执行 TimeLine：

TimeLine 提供了一个可视化的的时间表，直观地展现出了任务的执行顺序、抢

占情况、执行时间，为开发人员的分析工作带来了便利。

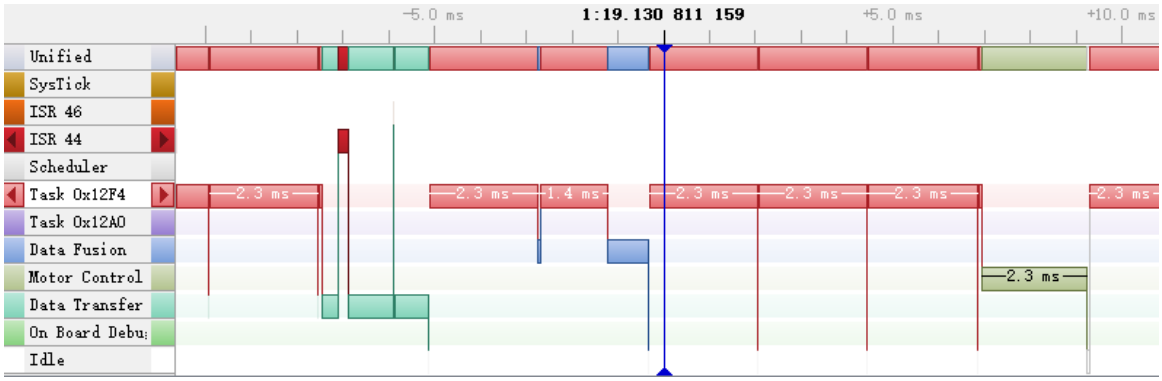


图 2-4 TimeLine

CPU 占用率：

通过柱状图的方式展现了每一个时刻 CPU 占用的情况，能够让开发者清楚地知道任务运行过程中每一时刻 CPU 资源的消耗情况。

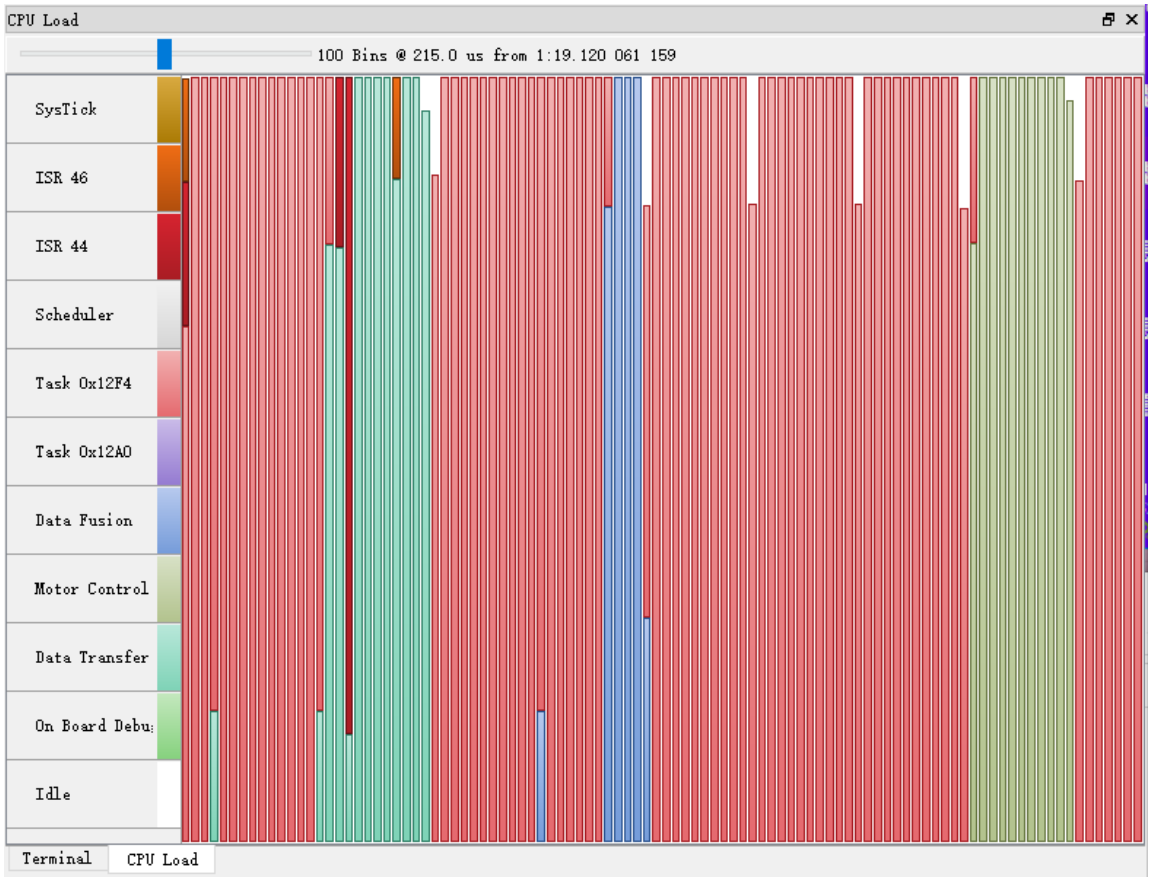


图 2-5 CPU 占用率

第三章 知识技能学习情况

3.1 中断问题学习

一、 中断向量表与重定位

异常处理的第一个步骤，就是找到异常向量表。

与传统 ARM 处理器向量表不同：传统 ARM 处理器向量表中存在跳转到相应处理的指令，而 CM 处理器向量表为异常处理的起始地址

向量表可以重定位，在 SBC->VTOR 在，值默认是 0，及向量表的偏移地址量为 0。在向量表偏移过程中我们也需要考虑对齐问题，因为 STM32F4 处理器采用了 Thumb 指令集，所以向量表偏移的时候需保证最后两位和内存空间对齐，此外所有异常发生时，其指令的最低为都是 1，不然会产生 HardFault 错误。

二、 中断响应过程中 STM32 自动完成的工作

ARM 架构 C 编译器遵循 AAPCS，C 函数能修改 R0-R3（参数）、R12（临时寄存器）、R14（LR）、和 PSR（“调用者保存寄存器”），如果要用 R4-R11，就应该把这些寄存器保存在栈空间中，在函数结束前恢复它们

对于调用者保存寄存器，如果要使用，在调用前，调用子程序的程序代码需要将这些寄存器内容保存到栈中，调用后不需要使用的可以不保存。

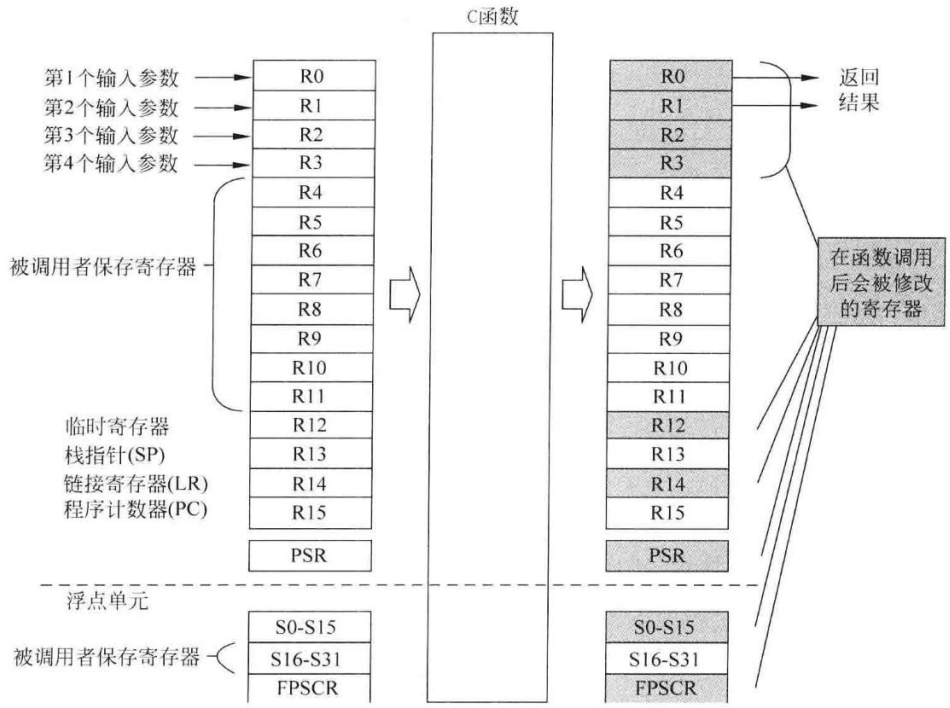


图 3-1 AAPCS 规定的函数调用中寄存器的使用

异常处理也是一种程序的切换，所以也要保存寄存器，不过这时是在异常入口处由处理器自动保存的调用者保存寄存器。

三、 EXC_RETURN

表 8.1 EXC_RETURN 的位域

位	描述	数 值
31:28	EXC_RETURN 指示	0xF
27:5	保留(全为 1)	0xEFFFFF(23 位都是 1)
4	栈帧类型	1(8 字)或 0(26 字)。当浮点单元不可用时总是为 1,在进入异常处理时,其会被置为 CONTROL 寄存器的 FPCA 位
3	返回模式	1(返回线程)或 0(返回处理)
2	返回栈	1(返回线程栈)或 0(返回主栈)
1	保留	0
0	保留	1

表 8.2 EXC_RETURN 的合法值

	浮点单元在中断前使用(FPCA=1)	浮点单元未在中断前使用(FPCA=0)
返回处理模式(总是使用主栈)	0xFFFFFFE1	0xFFFFFFF1
返回线程模式并在返回后使用主栈	0xFFFFFFE9	0xFFFFFFF9
返回处理模式并在返回后使用进程栈	0xFFFFFFED	0xFFFFFFFD

EXC_RETURN 的功能就是进行中断推出，根据使用 EXC_RETURN 值的不同，可以设置返回的时候回到哪个模式、使用哪个栈。

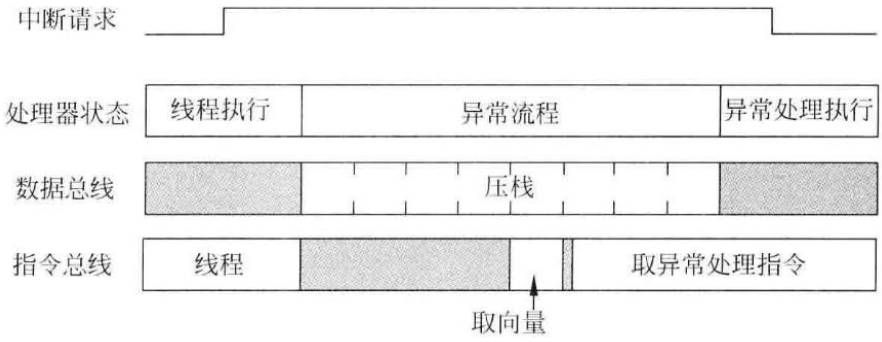


图 3-2 中断发生时总线状态

在中断请求到来时，会在数据总线自动执行压栈，压栈的同时指令总线自动从向量表取得中断向量的地址，然后进行中断处理。

下面介绍使用不同栈与不同模式的时候的异常栈帧：

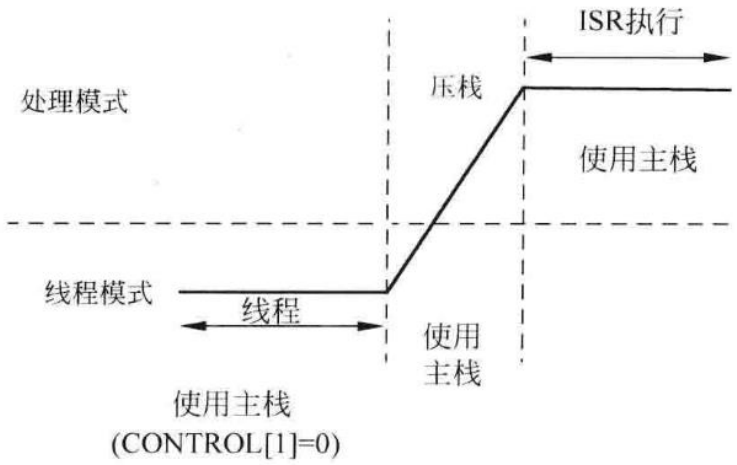


图 3-3 使用主栈的线程模式的异常栈帧

使用主栈的线程模式，发生中断时，处理器模式会切换到处理模式，并且自动继续使用主栈执行 ISR。

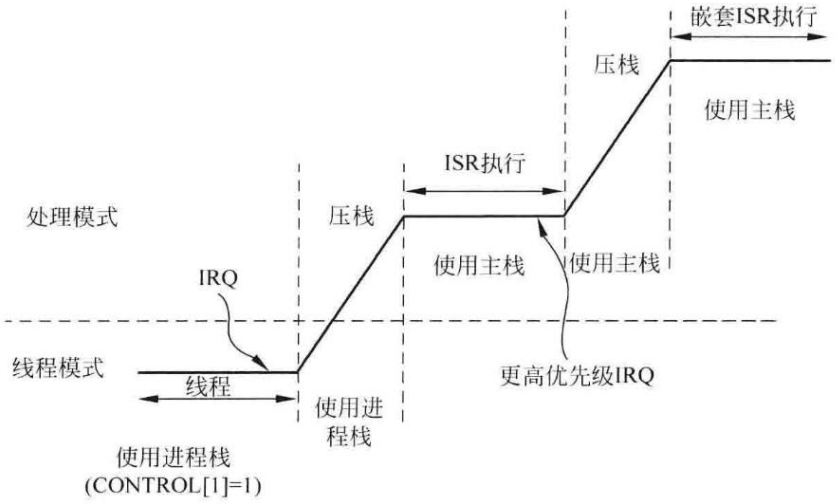


图 3-4 使用进程栈的线程模式的异常栈帧，以及使用主栈的嵌套中断压栈

使用该模式时，在进入中断后，会自动使用主栈，然后当更高优先级中断到来的时候仍然会继续使用主栈进行嵌套 ISR 的执行。

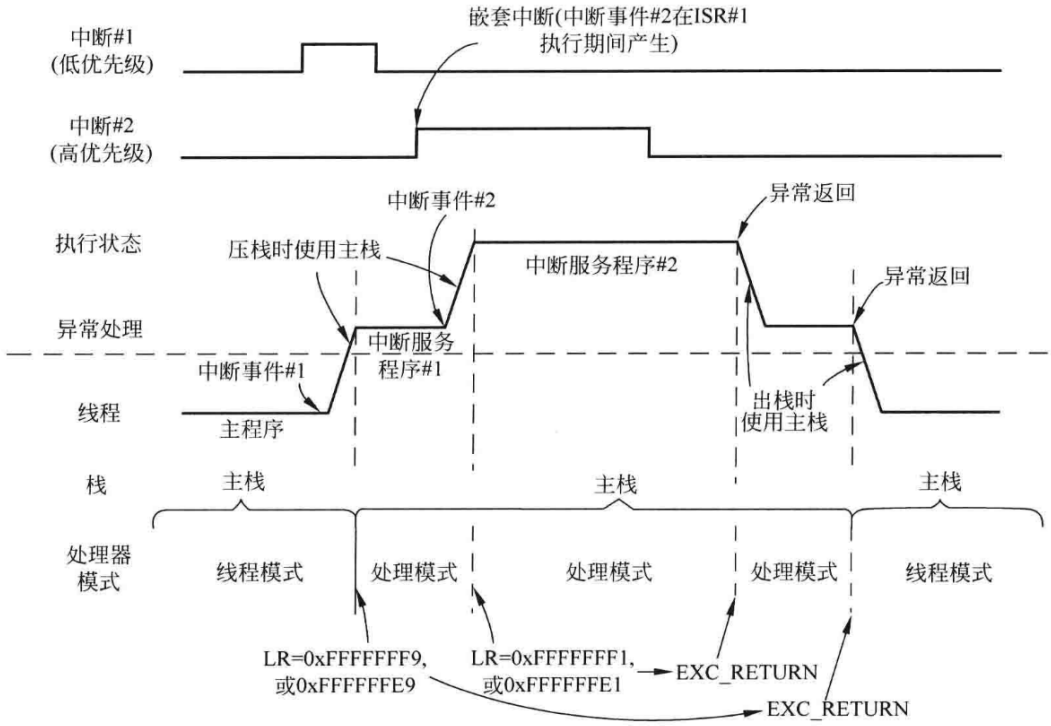


图 3-5 LR 在异常时被设置为 EXC_RETURN

上图显示出了在整个中断执行流程中，如果发生了中断嵌套，LR 寄存器中的

中断返回值将如何设置。具体操作再次不进行赘述。

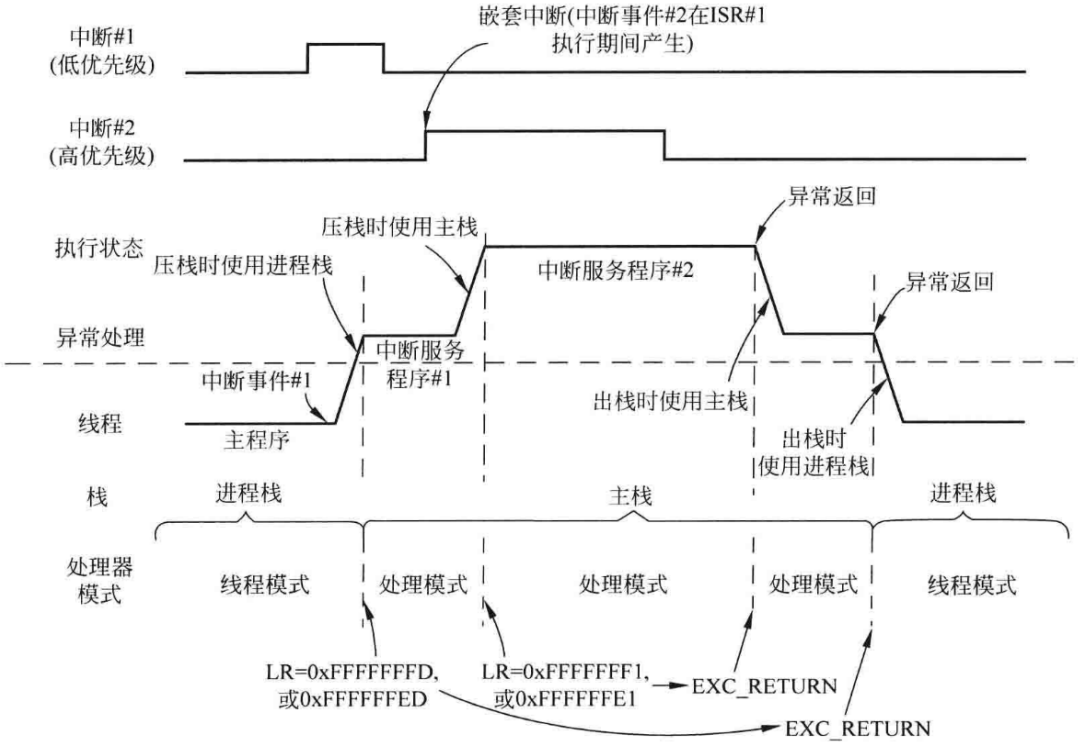


图 3-6　LR 在使用进程栈发生中断时的变化

下面介绍在中断发生时的一些特殊情况，比如说末尾连锁（又被称作“咬尾”）。这相当于操作系统在两个中断中间不恢复现场。

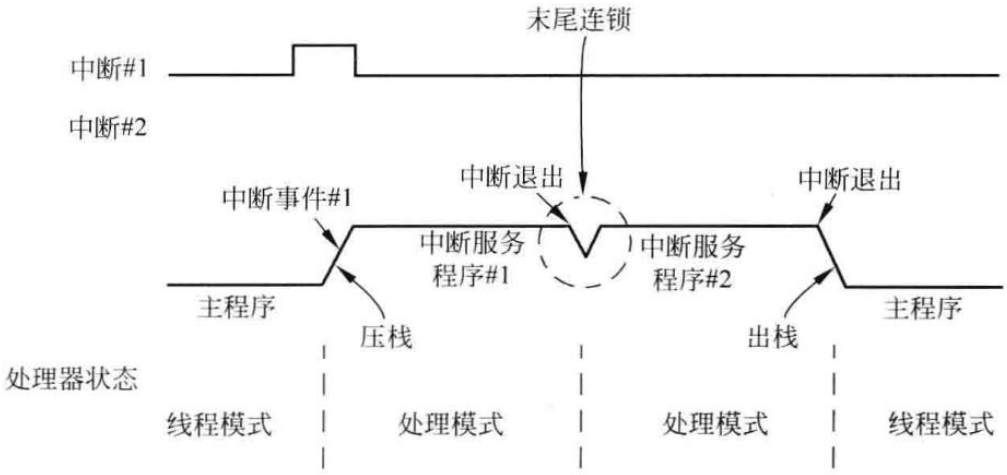


图 3-7 中断末尾连锁情况

3.2 内核资源互斥访问

一、 互斥与临界区

互斥：我们把一个时间段内只允许一个任务使用的资源称为临界资源。对临界资源访问的这种排他性就是互斥

临界区：每个任务访问共享资源的那段程序称为临界区（Critical Section）。在临界区不允许任务切换，如果在访问共享资源时进行任务切换，可能发生灾难性后果。因此，在进入临界区访问共享资源前，采用关中断，给调度器上锁或使用信号量的方法达到互斥的目的。

信号量：uC/OS II 采用信号量作为任务与任务间的同步和互斥机制。信号量(Semaphore)，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。

uC/OS II 中的信号量由两部分组成：一部分是信号量的计数值（16 位无符号整数），另一部分是等待该信号量的任务组成的任务等待队列。

uC/OS II 提供了三种信号量：

1. 用于任务同步的信号量。
2. 任务对共享资源互斥访问的互斥信号量。
3. 对系统多个资源管理的信号量。

二、 几种进入临界区的方式

代码 3-1 临界区代码

```
#if OS_CRITICAL_METHOD == 1
#define OS_ENTER_CRITICAL() {Cli();}
#define OS_EXIT_CRITICAL() {Sti();}
#endif


#if OS_CRITICAL_METHOD == 2
#define OS_ENTER_CRITICAL() {PushAndCil();}
```

```
#define OS_EXIT_CRITICAL() {Pop();}
#endif
#if OS_CRITICAL_METHOD == 3
#define OS_ENTER_CRITICAL() {cpu_sr = OS_CPU_SR_Save();}
#define OS_EXIT_CRITICAL() {OS_CPU_SR_Restore(cpu_sr);}
#endif
```

3.3 MakeFile 工程构建

一、 交叉编译工具链

ARM-GCC

一套交叉编译工具链家族，其命名规则统一为：arch [-vendor] [-os] [-(gnu)eabi]

arch：代表芯片的体系架构，比如 ARM，MIPS 等。

vendor：代表工具链的提供商。

os 代表目标开发板所使用的操作系统。

eabi：代表 Embedded Application Binary Interface，即嵌入式应用二进制接口。

arm-none-eabi-gcc(ARM architecture, no vendor, not target an operating system, complies with the ARM EABI)主要用于编译 ARM 架构的裸机系统（包括 ARM Linux 的 Boot 和 Kernel，不适用编译 Linux 应用），一般适合 ARM7、Cortex-M 和 Cortex-R 内核等芯片使用，不支持那些跟操作系统关系密切的函数。除此之外，该编译器在底层使用了 newlib 这个专用于嵌入式系统的 C 库。

了解完原理后，我们需要设置环境变量安装 32 位库 ia32-libs 并且验证是否安装成功。安装成功后重启 reboot

二、 程序代码编译过程

预处理：源代码文件（.c）和相关头文件（.h）被预处理器 cpp 预编译成.i 文件，主要处理的是源代码中以#开头的预编译指令。

编译：把预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后被编译器 cc1 编译成相应的汇编代码文件(.s)

汇编：将汇编代码转变成机器可执行的命令，生成目标文件.o，汇编器 as 根据汇编指令和机器指令的对照表一一翻译即可完成。

链接：链接就是链接器 ld 将哥哥目标文件组装在一起，解决符号以来，库依赖关系，并生成可执行文件。

三、 MakeFile 基础

1. 定义变量：为编译链接服务，使编译链接过程简洁化、自动化。

最终生成的目标文件

可选设置：生成中间文件存放位置、是否 DEBUG、优化等级

源文件：全部参与编译的.c 和.s 的文件

头文件：全部参与编译的.h 文件

目标文件：全部需要生成的.o 文件

编译选项 CFLAGS 和 ASFLAGS

链接 LDFLAGS

2. 编译链接依赖关系

最终需要生成三个文件：.elf .hex .bin

通过.elf 生成 .hex .bin

通过链接.o 生成.elf

通过编译.c .h 生成部分.o ；通过汇编.s 生成部分.o

四、 链接文件与链接脚本

链接脚本 LinkScript

主要功能是描述如何把输入文件中的节（sections）映射到输出文件中，并控制输出文件的存储布局。

当然，在大多数情况下我们都不会注意到链接脚本的存在，主要原因在于链接器在我们没有指定特定链接脚本的时候，会使用一个默认缺省的脚本。

3.4 任务划分问题

一、 应用层任务的划分问题

基于 DARTS 的任务划分原则

- I/O 依赖性

如果变换依赖于 IO，速度受限 IO，可独立成任务。

在系统中创建与 IO 设备数目相当的 I/O 任务。

IO 任务只实现与设备相关的代码。

IO 任务的执行只受限于 IO 设备的速度，而不是处理器在任务中分离设备相关性。

- 功能的时间关键性

具有时间关键性的功能应当分离处理出来，成为一个独立的任务，并且赋予这些任务较高的优先级，以满足系统对时间的要求。

- 计算需求

计算量大的功能在运行时势必会占用 CPU 很多时间，应当让它们单独成为一个任务。

为了保证其他费时少的任务得到优先运行，应该赋予计算量大的任务以较低优先级运行，这样允许它能被高优先级的任务抢占。

多个计算任务可安排成同优先级，按时间片循环轮转。

- 功能内聚

系统中各紧密相关的功能,不适合划分为独立的任务，应该把这些逻辑上或数据上紧密相关的功能合成一个任务，使各个功能共享资源或相同事件的驱动。

把每个变换都作为同一任务中一个个独立的模块,不仅保证了模块级的功能内聚，也保证了任务级的功能内聚。

- 时间内聚

将同一时间内完成的各功能形成一个任务，即使这些功能是不相关的。

功能组的各功能是由相同的外部事件驱动的（如时钟等)，这样每次任务接收到一个事件，它们都可以同时执行。

由于减少了任务调度及切换的次数，减少了系统的开销。

- 功能的周期执行

将在相同周期内执行的各个功能组成一个任务，使运行频率越高的任务赋予越高的优先级。

频率高的任务赋予高优先级。

第四章 分工协作与交流情况

1. 小组成员：3 人

   组长：徐洋

   组员：余子潇，唐昊哲

2. 每人完成的详细任务

   徐洋：应用程序调试，上下文切换与临界区

   余子潇：操作系统移植，事件与时钟驱动

   唐昊哲：操作系统调试，系统启动代码，SystemView

3. 交流情况：

在整个项目开发和测试阶段，我们小组主要依靠程序和文档实时共享的模式进行开发。利用 NAS 网络附属存储管理所有的程序，图片以及视频。组内人员各司其职分工明确，当遇到问题时，一般采用网上讨论+线下实践，学习开发文档以及网络资源后，再当面交流沟通的方式，高效完成开发任务。

参考文献

[1] 廖勇 杨霞主编.嵌入式操作系统[J].2017.1

[2] STM32F4xx 中文参考手册[J].意法半导体

[3] STM32 固件库使用手册[J].意法半导体

[4] STM32F4 开发指南-寄存器版本[J].正点原子

[5] 零死角玩转 STM32—F103[J].野火

[6] Joseph Yiu 著 吴常玉 曹孟娟 王丽红译.ARM Cortex-M3 与 Cortex-M4 权威指南.2015

[7] 汤小丹 梁红兵 汤子瀛.计算机操作系统（第 4 版）2014.5

致谢

本报告的工作是在我们敬爱的指导教师廖勇老师的悉心指导下完成的。廖老师在整个综合课程设计中主动帮助我们分析问题。组织大家共同讨论，制定任务，让同学们互相监督，共同进步。遇到无法解决的问题时廖老师都一一耐心讲解。对整个课程设计的指导从知识讲授、文档撰写到答辩技巧无微不至。在此，谨向老师表达我们衷心的感谢和崇高的敬意！

另外，感谢学院给予我们这样的机会，能够独立的完成一个课程设计，并在这个过程当中，为我们提供教师资源以及学习资源的帮助，使我们能够学以致用，真正地将课堂中学习到的知识转为实际的计算机系统结构能力，增强了我们的实践操作水平以及合作交流分工能力。

还要感谢在课程设计完成的过程中帮助过我们的老师、前辈和同学们。感谢你们的提出的宝贵经验和建议，让我们的工作少走弯路，没有你们的耐心指导，就没有项目的顺利完成。

最后再次对帮助给我们的所有人表示由衷的感谢。