RTOS 内核实现

--基于 Cortex-M3

作者：江小鉴

前言

在当今时代，实时操作系统（Real Time Operate System，简称RTOS）的种类繁多，国际上流行的包括VxWorks、QNX、FreeRTOS、μC/OS-II等，而国内市场上也涌现出了RT-Thread和DeltaOS等优秀选项，每个RTOS都拥有其独特的优点和适用范围。

例如，FreeRTOS是一个开源且免费的实时操作系统内核，它以可裁剪性著称，适用于各类嵌入式系统。该内核以其微小的内存占用和高效的实时性能而闻名，支持多种任务调度机制，包括抢占式调度和时间片轮转调度。μC/OS-II则是一种以小巧内核和简单性为特点的抢占式实时操作系统，它以快速响应和高可靠性著称。在国内市场上，RT-Thread也是一个广泛使用的嵌入式操作系统，它以其出色的兼容性和可扩展性受到青睐。

每个RTOS都拥有其独特的优点和特性，我确信你也和我一样对这些RTOS的精妙设计感到惊叹，然而，RTOS的多样性也让我在选择时感到困惑，每次深入阅读这些源代码，我都感觉自己像是陷入了无底深渊，难以完全掌握它们的精髓。因此，我决定创建一个属于自己的RTOS，包括任务管理、内存管理和定时管理等核心功能。本文将详细记录我构建这个RTOS的过程，以及其中包含的设计理念。在本文探讨的过程中，你将能够掌握RTOS的基本原理，并能成功编写一个RTOS。

本文不会深入探讨所有引用的原理，因为过于详尽的解释可能会使文章陷入仅限于解释原理的循环之中，在多数情况下，我会在文中提供建议参阅的其他资料。尽管本文在构建RTOS的过程中提供了详尽的步骤，但我建议读者还是应具备一定的C语言和单片机基础，否则文章中的某些内容可能会难以理解。

本文遵循先理论后实践的原则，我将对Cortex-M3 的相关资源进行解读，并通过这些资源构建自编写的RTOS的理论基础，随后我将从理论出发，着手编写代码。我始终秉承以功能实现为首要目标的原则，在编写RTOS的过程中，由于本人知识与经验的局限性，某些原理可能有所疏漏，部分代码可能未完全遵循既定规范或存在设计上的不足，敬请谅解。

目录

前言 1

第一章 RTOS是什么？ 1

1.1 RTOS的概念 1

1.2 为什么要使用RTOS 1

1.3 一些建议 2

第二章 Cortex-M3 内核简介 4

2.1 运行模式和特权状态 4

2.2 异常处理 5

2.2.1 SVCall异常 7

2.2.2 PendSV异常 7

2.2.3SysTick异常 8

2.3 寄存器 8

2.3.1 R13 堆栈指针SP 9

2.3.2 R14 链接寄存器LR 9

2.3.3 R15 程序计寄存器PC 9

2.3.4 xPSR寄存器 9

2.3.5 PRIMAS寄存器 11

2.3.6 FAULTMASK寄存器 11

2.3.7 BASEPRI寄存器 11

2.3.8 CONTROL寄存器 11

小结 11

第三章 工程建立初体验 13

3.1 环境搭建 13

3.1.1 STM32CubeMX 13

3.1.2 ST-LINK驱动 14

3.1.3 Keil 14

3.1.4 CH340 驱动 15

3.2 Stm32CubeMX工程建立 15

3.3 打开Keil工程并编写LED闪烁程序 19

小结 21
第四章 任务理论构建 22
4.1 任务的概念 22
4.2 任务切换原理 22
4.3 任务的状态 23
4.4 任务的堆栈 23
4.5 时间片轮转 23
4.6 任务链表设计 24
小结 25
第五章 流程与规则 26
5.1 C语言编译过程 26
5.2 启动流程 27
5.3 ATPCS标准 29
5.4 常用的汇编指令 30
5.5 自动入栈与出栈 31
小结 32
第六章 内核编写 33
6.1 头文件定义 33
6.2 任务创建 36
6.3 任务切换 40
6.4 系统时钟事件触发 45
6.5 任务就绪 48
6.6 内核第一次运行 48
6.7 任务延时 52
6.8 任务暂停与删除 52
6.9 外部使用 53
小结 54
第七章 任务抢占式 55
7.1 旧任务管理流程分析 55

7.2 新任务管理流程设计 56
7.3 头文件代码修改 57
7.4 抽象链表操作 59
7.5 任务结束事件与主动切换事件 66
7.6 任务结束处理 69
7.7 修改任务创建函数 71
7.8 修改任务暂停和删除函数 73
7.9 修改异常处理函数 75
7.10 修改任务延时函数 77
7.11 其他修改 78
7.12 文件分离 80
小结 81
第八章 定时任务 82
8.1 定时任务的特点 82
8.2 定时任务函数编写 82
小结 83
第九章 内存管理 85
9.1 内存管理设计 85
9.2 内存申请 86
9.3 内存释放 89
9.4 内存初始化 91
9.5 系统栈与任务栈分离 92
小结 92
第十章 Hello world！实现 93
10.1 串口配置 93
10.2 打印实现 94
小结 100
第十一章 内核休眠 101
11.1 内核休眠原理 101

11.2 休眠实现 102
第十二章 CPU使用率统计 104
12.1 统计原理 104
12.2 CPU使用率实现 105
第十三章 优化内存管理 113
13.1 优化原理 113
13.2 优化内存管理实现 119
最后 123

第一章 RTOS是什么？

1.1 RTOS的概念

RTOS是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统做出快速响应，调度一切可利用的资源完成实时任务的一种操作系统。

RTOS是专门为实时任务设计，它的任务调度算法和中断处理机制都是为了确保实时性。RTOS的特点是响应速度快、可靠性高、稳定性好、实时性强等。RTOS广泛应用于各种领域，如航空航天、军事、汽车、医疗、工业控制等。在这些领域中，系统需要对外部事件做出快速响应，以确保安全和效率。

例如，在航空航天领域中，飞机控制系统需要实时响应飞行员的操作指令和外部环境变化，以确保飞机的稳定和安全。在汽车领域中，刹车系统需要实时响应驾驶员的刹车操作，以缩短制动距离并避免事故发生。在医疗领域中，呼吸机需要实时监测患者的呼吸状态，并根据需要调整呼吸频率和压力，以确保患者的生命安全。

除了响应速度快之外，RTOS还具有高度可靠性和稳定性。由于RTOS的任务调度算法和中断处理机制都是针对实时性设计的，因此它们能够保证任务的及时执行和系统的稳定运行。此外，RTOS还提供了丰富的功能模块和接口，以满足不同应用场景的需求。

总之，RTOS是一种非常重要的操作系统，它为各种实时应用提供了可靠的基础支持。随着技术的不断发展和应用的不断扩展，RTOS将继续发挥重要作用。

1.2 为什么要使用RTOS

在功能受限的小型系统中，尽管裸机程序已经足以支撑系统的基本运作，但在必须处理多项任务的场景下，RTOS能够更加高效地进行任务管理和调度。RTOS能够根据任务的具体需求来执行任务调度，允许为每个任务分配优先级，从而确保关键任务能够及时得到执行。鉴于嵌入式产品对资源和功耗的高度敏感

性，RTOS设计时考虑了可裁剪性、低功耗和低资源占用率等特性。RTOS还强化了系统的实时性，确保所有任务均能在既定的时间限制内完成。

综上所述，实时操作系统（RTOS）在提升系统实时性、稳定性和可靠性方面扮演了至关重要的角色，使得嵌入式系统能够更好地适应现代工业和日常生活的需求。

1.3 一些建议

以下是我个人在学习过程中积累的一些经验和建议，或许可以称之为忠告。在编写RTOS过程中，对于某些不太关键的原理，你不必过于深入地去理解每一个细节。然而，对于那些核心的原理，你必须彻底弄清楚它们的底层逻辑，任何现象背后都有其存在的原因。有时候，你必须静下心来，一步一个脚印地去探索，才能真正找到问题的根源，如果你的学习只是浮光掠影，那么你最终只能复制别人的东西，而无法创造出真正属于自己的作品。

坦率地说，在编写RTOS的过程中，我参考了大量相关的资料和文献，但我始终保持独立思考，不被这些资料所左右。我追求的是理解“为什么这样做”，而不仅仅是“我只能这样做”。通过深入探究每一个必要的原理和概念，我能够更好地掌握它们，并在此基础上进行创新和改进。在这个过程中，我深刻体会到理论与实践相结合的重要性，理论知识为我们提供了方向和框架，但真正让知识活起来的，是将其应用于实际项目中。因此，我鼓励自己在理解理论的同时，不断动手实践，哪怕是编写一些小的功能模块或者模拟实验。这种“做中学”的方式，不仅加深了我对RTOS运行机制的理解，也让我在遇到问题时，能够更迅速地从多个角度思考解决方案。

在学习的过程中，至关重要的是保持勇气，不畏惧编写代码，也不畏惧投入时间。有时，你可能只是渴望休息，并非打算放弃。然而，许多人一旦在学习中感到疲惫，便会选择放弃。学习一旦中断，便可能让人感到困惑，从而陷入恶性循环，放弃得越多，就越难以继续，最终渐行渐远。确实，每个人都会有这样的时刻，我也不例外，每当这种感觉出现，我便意识到自己已经触及了当前的极限，我可能需要休息，或者重新审视我的方法。休息之后，我会重新整理思路，着手解决问题。对我来说，每次重新开始都是一个艰难的过程，我也会感到恐惧，头

脑中充满那种难以言表的痛苦，以及来自内心的抗拒。这足以让人畏缩不前，但同时，这也正是其乐趣之所在，因为当你再次踏上这段旅程时，沿途的风景定会有所不同。

持续学习是成为优秀开发者的必由之路，技术领域日新月异，唯有不断挑战自我，勇敢地走出舒适区，我们才能与时代的脉搏同步。编写优质代码不仅是一项技术上的考验，更是一次个人成长和自我提升的旅程。我相信，每个人都有能力创作出更加完美的作品。

第二章 Cortex-M3 内核简介

Cortex-M3 是ARM公司推出的一款专为微控制器市场设计的处理器核心，基于ARMv7-M架构。该核心的指令集构成了Cortex-M3 处理器运行软件的基础，它规定了一系列机器指令，使开发者能够编写程序来控制硬件。

若要开发自己的RTOS，则必须深入研究处理器内核。本文不涉及Cortex-M3 的其他特性，更多详细信息请参阅《Cortex-M3 权威指南》，本文将重点阐述实现RTOS所需的重要资源。

2.1 运行模式和特权状态

Cortex-M3 处理器支持两种处理器的运行模式和两种特权状态。这两种处理器运行模式分别是handler模式和thread模式。handler模式与thread模式的主要区别在于它们分别用于处理异常代码和正常代码。具体而言，异常代码必须在handler模式下执行，以确保系统的稳定性和安全性。

在特权状态方面，Cortex-M3 处理器拥有特权级和用户级两种级别。当处理器处于特权级时，它将拥有对所有寄存器的访问权限以及执行所有指令的权限。这种权限级别通常用于执行关键的系统操作，如中断处理、任务调度等。然而，在用户级，处理器的权限则受到限制，无法访问一些关键的寄存器和执行一些敏感的指令。这种设计有助于增强系统的安全性和稳定性，防止恶意代码或用户误操作对系统造成破坏，Cortex-M3 处理器的运行模式和特权状态设计为其在各种应用场景中提供了灵活性和安全性。通过合理利用这些特性，可以构建出更加高效、稳定、安全的嵌入式系统。

两种处理器的运行模式和特权状态是相互交错的。在特权级别下，处理器可以运行于handler模式或thread模式。然而，在用户级别下，处理器仅限于运行于thread模式，无法采用handler模式。

表 1 运行模式与特权状态

	特权级	用户级
handler模式	√	×

thread模式	√	√

在开发裸机代码的过程中，我们通常保持在特权模式下。然而，为了构建一个RTOS，我们需要结合使用特权模式与用户模式，以增强系统的健壮性。在特权模式下，我们可以通过调整特定寄存器轻松切换到用户模式。但是，一旦切换到用户模式，系统就不能随意返回特权模式。在用户模式中，系统必须通过特定的机制来通知内核，这种机制就是触发异常。当异常被触发时，Cortex-M3 处理器会进入handler模式，在此模式下，处理器的操作权限自动提升至特权级，从而实现了从用户模式到特权模式的转换。

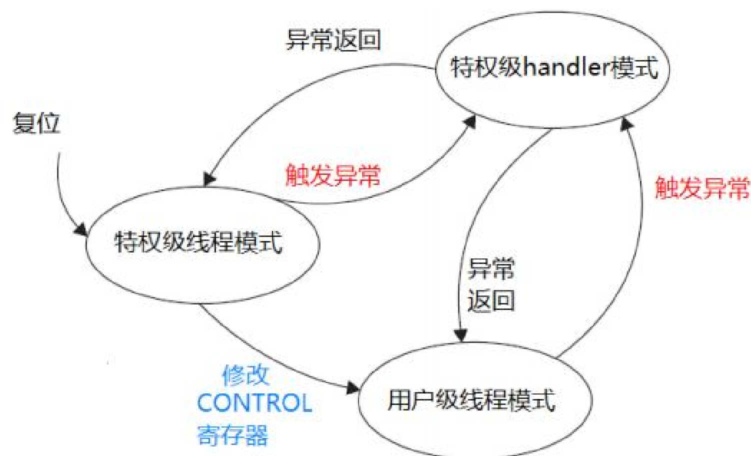

图 1 运行模式与特权状态相互转换

2.2 异常处理

什么是异常？在探讨异常时，我们不得不提及中断。通常情况下，一旦出现异常，中断便接踵而至（接下来所讨论的异常均涵盖了中断的概念）。举个例子，当你正与朋友品茶畅谈时，突然电话铃声响起。你查看来电显示，决定是否接听，在这种情境下，电话铃声响起即被视为一个异常，而你选择接听电话则相当于经历了一次中断，接完电话后，你再回到之前的对话中，这就意味着中断已经结束，然而，如果来电显示的是一个不重要的号码，你选择不接听，那么这就仅仅是一个异常，而没有引起中断。

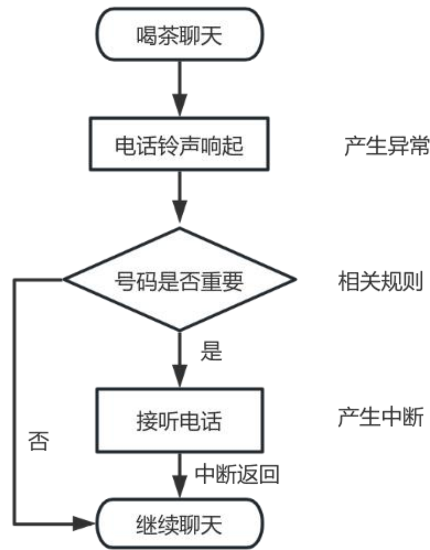

图 2 异常响应流程

在Cortex-M3 处理器中，当发生异常这可能是由于硬件故障、用户程序请求中断或定时器溢出等原因引起的，内核将依据既定规则以及异常的优先级来决定是否触发一个中断以响应该异常。

每个异常均被赋予一个特定的编号和优先级。编号范围从 1 至 15 的异常属于系统异常，这些由ARM公司预定义。而编号从 16 至 255 的异常则为外部中断，通常由半导体制造商根据具体需求进行定制。

表 2 Cortex-M3 主要关注的异常编号表

编号	类型	优先级	简介
0	N/A	N/A	没有异常
...	...	...	...
11	SVCall	可编程	系统服务调用
...	...	...	...
14	PendSV	可编程	可悬挂请求
15	SysTick	可编程	系统滴答定时器
...	...	...	...

除了几个特定的异常之外，大多数异常允许用户自定义其优先级。在探讨系统异常时，我们将重点放在SVCall异常、PendSV异常和SysTick异常上，而其他异常在此暂不涉及。

2.2.1 SVCall异常

SVCall指令用于执行系统服务调用，通常我们希望程序能够主动引发异常。Cortex-M3 处理器提供了SVCall异常处理机制，允许用户程序在运行于用户模式时，通过主动触发SVCall异常来获得特权模式权限。这样，程序便能执行那些仅限于特权模式的操作。通过这种方式，用户模式与特权模式得以明确区分，用户程序因此能够避免陷入权限管理的复杂性。此外，这也是用户级程序主动转变为特权级的唯一途径。

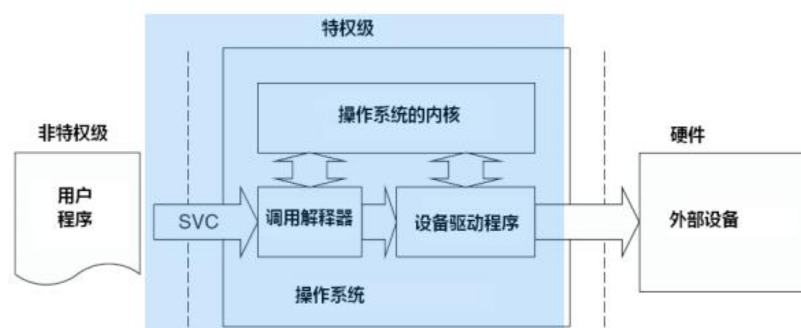

图 3 SVCall异常调用

2.2.2 PendSV异常

PendSV是一种待处理的软件中断，用于在内核中标记一个异常，随后产生中断处理异常，通常情况下，在程序正在其他中断处理程序中时，内核会等待当前中断执行完成后响应这个PendSV异常。PendSV异常主要用于任务切换，当用户程序正在执行时，如果发生另一个异常，系统会进入相应的异常中断处理程序，如果在中断程序中执行任务切换，将会中断当前的中断处理程序，导致任务切换到下一个任务，而当前中断处理尚未完成，这种强制性的任务切换可能会引发不可预测的后果，在程序设计中这是绝对不能被允许的！！！通过使用PendSV异常，可以避免这种情况，在中断处理过程中触发PendSV异常时，内核不会立即产生PendSV中断，而是会等待当前中断处理完全结束后，再进入PendSV中断进行处理。

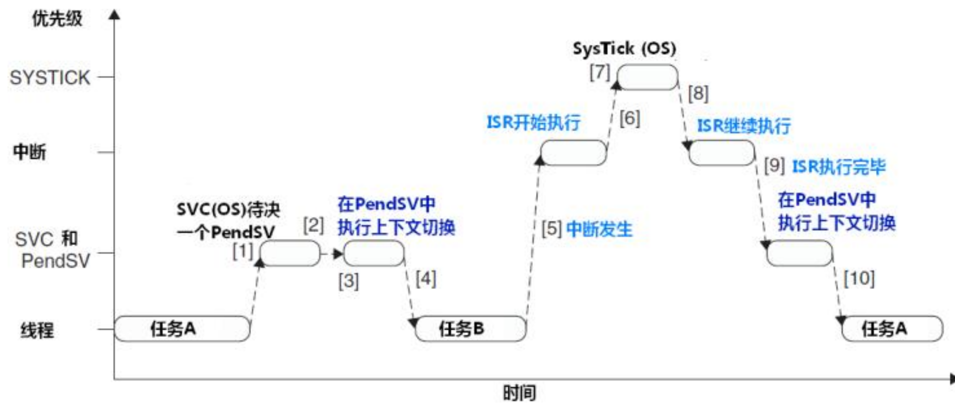

图 4 PendSV异常响应

2.2.3SysTick异常

SysTick是系统滴答定时器，负责周期性地触发一个溢出异常。这一异常通常被用于系统时钟的计数以及挂起PendSV异常。

2.3 寄存器

Cortex-M3 配备了由R0 至R15 组成的寄存器组。其中，R0 至R12 作为通用寄存器，R13 作为堆栈指针（SP），R14 作为链接寄存器（LR），而R15 则是程序计数器（PC）。

表 3 寄存器集合

寄存器名称	类型
R0-R12	通用寄存器
R13(SP)	堆栈指针寄存器
R14(LR)	链接寄存器
R15(PC)	程序计数器
xPSR	状态寄存器
PRIMASK、FAULTMASK、BASEPRI	中断屏蔽寄存器
CONTROL	特权状态控制寄存器

此外，Cortex-M3 还包含多个特殊功能寄存器：xPSR、PRIMASK、FAULTMASK、BASEPRI和CONTROL。xPSR用作状态寄存器，而PRIMASK、FAULTMASK和BASEPRI是中断屏蔽寄存器，CONTROL寄存器负责定义特权状

态。

2.3.1 R13 堆栈指针SP

R13 是堆栈的指针，始终指向栈底。在Cortex-M3 中，R13 具备两个堆栈指针功能，即主堆栈指针（MSP）和进程堆栈指针（PSP）。这是ARM架构的一种独特设计，确保在同一时刻只能激活其中一个堆栈指针。MSP作为复位后的默认堆栈指针，理论上在程序执行过程中持续使用MSP是可行的。在编写裸机程序时，MSP通常是首选的堆栈指针。而PSP则是由用户根据需求进行切换的，它更多地被应用于用户级代码的执行中。

为何ARM架构会引入MSP和PSP这两种堆栈指针呢？实际上，使用单一堆栈指针足以应对大多数情况，但ARM的设计理念在于增强安全性。通过区分特权级和用户级的堆栈指针，将特权级堆栈指针置于受保护的内存区域，而用户级堆栈指针则位于另一独立区域，这种设计可以有效防止用户级程序的潜在错误操作导致整个系统的不稳定。

2.3.2 R14 链接寄存器LR

R14 寄存器主要用于存储程序返回地址。在发生异常时，CPU会将当前程序即将执行的下一条指令地址保存至R14 中，以便在异常处理完毕后能够顺利返回到原程序继续执行。

2.3.3 R15 程序计寄存器PC

R15 寄存器始终指向当前程序的执行地址。通过修改R15 的值，可以有效地改变程序的执行流程，从而实现跳转功能。

2.3.4 xPSR寄存器

xPSR寄存器是程序状态寄存器，顾名思义，它记录了程序运行过程中的所有状态。它由三个子寄存器组成，分别是应用程序APSR、中断号IPSR和执行EPSR，这三个都可以单独访问，也可以一次性访问。

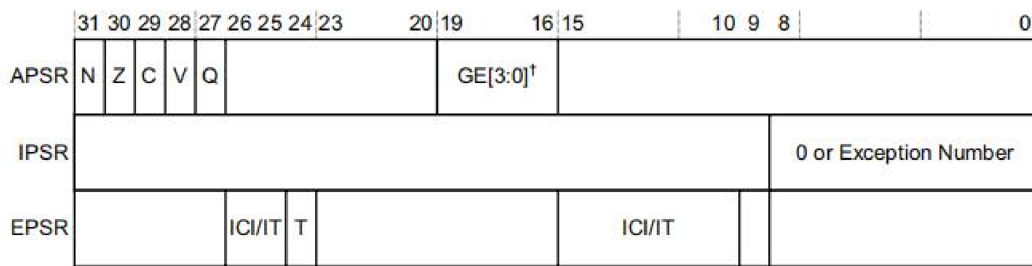

图 5 xPSR状态寄存器

APSR：APSR的标志标志位N、Z、C、V和Q，处理器使用这些标志来评估IT和条件分支指令中的条件执行。GE位与DSP扩展有关，Cortex-M3 没有DSP，这里不关注。

（1）N位（负数标志）：当运算结果为负数时，N位被设置为 1；否则为 0。它用于指示运算结果的符号。

（2）Z位（零标志）：当运算结果为零时，Z位被设置为 1；否则为 0。它用于检测运算结果是否为零。

（3）C位（进位标志）：在加法或减法运算中，如果产生了从低位向高位的进位或借位，C位被设置为 1；否则为 0。它用于指示运算过程中的进位或借位情况。

（4）V位（溢出标志）：在运算过程中，如果结果超出了寄存器能够表示的范围，V位被设置为 1；否则为 0。它用于指示运算是否发生了溢出。

（5）Q位（饱和标志）：在某些特定的运算中，如饱和算术运算，Q位用于指示运算结果是否达到了饱和状态。

（6）GE[3:0]：与DSP扩展相关。

IPSR：处理器在异常的进入和退出阶段会自动写入IPSR（中断程序状态寄存器）。软件开发者可以利用MRS指令来读取IPSR中的信息，然而，处理器会忽略任何通过MSR指令尝试对IPSR进行的写入操作。IPSR中的“Exception Number”字段具有以下定义：在thread模式下，该字段的值被设定为 0。而在handler模式下，它会保存当前正在处理的异常的唯一编号。

EPSR：ICI/IT是中断继续指令，IF-THEN指令状态位用于条件执行。T位将其设置为 1 以指示处理器执行Thumb指令，此在Cortex-M3 中此位必须为 1，否则将会导致错误。

总的来说，xPSR寄存器作为程序状态寄存器，在Cortex-M3 内核中扮演着至

关重要的角色。它记录了程序运行过程中的各种状态信息，包括运算结果的符号、是否为零、进位或借位情况、是否溢出等关键信息。这些信息为程序的正确执行提供了有力保障。

2.3.5 PRIMAS寄存器

这是一个仅具备单一位的寄存器，当该位被设置为 1，它将关闭所有可屏蔽的异常，仅保留非屏蔽中断（NMI）和硬故障（hard fault）的响应能力。其默认值为 0，这表示中断未被关闭。

2.3.6 FAULTMASK寄存器

这也是一个仅具备单一位的寄存器，当该位被设置为 1 时，唯有非屏蔽中断（NMI）能够得到响应，而所有其他类型的异常，包括硬件故障，都将被忽略。该寄存器的默认值为 0，意味着异常处理是开启状态。

2.3.7 BASEPRI寄存器

该寄存器最多有 9 位（由表示优先级的位数决定），它设定了屏蔽优先级的阈值，一旦设置为特定值，所有优先级编号大于或等于该值的中断将被禁用（优先级编号越高，表示优先级越低）。然而，如果设置为 0，则不会禁用任何中断，0 也是默认值。

2.3.8 CONTROL寄存器

CONTROL寄存器包含两个位。其中，CONTROL[0]位用于确定当前的特权级别：当该位设置为 0 时，表示选择特权模式；而当该位为 1 时，则表示处于用户模式。另一方面，CONTROL[1]位用于指定当前使用的堆栈指针：若该位为 0，则使用主堆栈指针（MSP）；若该位为 1，则使用进程堆栈指针（PSP）。

CONTROL寄存器通常在系统任务上下文切换过程中被用来配置用户程序以使用程序状态字（PSP）并切换至用户模式。

小结

本章简洁地概述了Cortex-M3 内核的运行模式、特权状态以及寄存器的功能。

特别地，xPSR寄存器主要用于追踪程序执行过程中的状态信息。PRIMASK寄存器的作用是控制所有可屏蔽中断，但不包括不可屏蔽中断（NMI）。FAULTMASK寄存器则用于管理所有硬件故障中断。BASEPRI寄存器设定了一个优先级阈值，仅允许优先级高于该阈值的中断被处理器响应。最后，CONTROL寄存器决定了处理器当前是处于特权模式还是用户模式。

第三章 工程建立初体验

我来对即将编写的RTOS命名为jdos吧，寓意为简单OS，旨在通过简洁的设计实现RTOS的核心功能。

本项目采用STM32F103RCT6 型号微控制器，STM32F103RCT6 是ST意法半导体公司生产的STM32 系列微控制器之一，它属于Cortex-M3 核心的高性能微控制器。为了更快的实现jdos功能，我不会从零开始构建项目——即不会从没有任何库文件的状态出发。相反，我将利用STM32CubeMX工具来生成Keil工程，这样可以快速生成和编译代码。STM32CubeMX可以从ST官方网站下载，而Keil软件则可以从Keil官方网站获取。我将尽量减少其他外部配置文件对编写jdos的影响，当然，如果你有能力探索其他环境也是可以的。以下是我的开发环境：

计算机系统：win10

STM32CubeMX：6.11.1

Keil：5.28

调试器：ST-LINK

3.1 环境搭建

本环境配置说明仅涵盖通用设置，若存在特定需求，用户可根据自身情况进行个性化配置，无需逐一遵循本文所述。

本工程的地址：https://gitee.com/jiang-xiaojian/jdos

3.1.1 STM32CubeMX

STM32CubeMX是ST意法半导体公司提供的一款图形化配置工具，它专门用于STM32 系列微控制器（MCU）和微处理器（MPU）的配置，并且能够生成相应的初始化C代码。这个工具的主要目的是简化开发者的工作，通过图形化界面来配置硬件参数，从而减少手动编写配置代码的工作量，提高开发效率。

软件下载地址：https://www.st.com/en/development-tools/stm32cubemx.html

Get Software

	Part Number	General Description	Latest version	Download	All versions
+	STM32CubeMX-Lin	STM32Cube init code generator for Linux	6.12.1	Get latest	Select version
+	STM32CubeMX-Mac	STM32Cube init code generator for macOS	6.12.1	Get latest	Select version
+	STM32CubeMX-Win	STM32Cube init code generator for Windows	6.12.1	Get latest	Select version

图 6 STM32CubeMX下载

选择你环境对应的版本，截止到目前，官网的最新版本为6.12.1，下载后直接安装即可。

3.1.2 ST-LINK驱动

ST-LINK是一种专门设计用于支持意法半导体（STMicroelectronics）公司生产的STM8 和STM32 系列微控制器芯片的仿真器。这种仿真器为开发者提供了一个强大的工具，使他们能够进行高效的调试和编程工作。ST-LINK仿真器具备多种功能，包括但不限于实时调试、程序下载、内存读写以及性能分析等。它通过USB接口与计算机连接，操作简便，支持多种开发环境，如IAR EWSTM8、Keil MDK-ARM和ST Visual Develop等。ST-LINK的广泛兼容性和高效性能使其成为STM8 和STM32 系列芯片开发者的理想选择。

驱动下载地址：https://www.st.com.cn/zh/development-tools/stsw-link009.html

下载完成后安装即可。

获取软件

	产品型号	ECCN (US)	下载
+	STSW-LINK009	EAR99	获取最新版本

图 7 ST-LINK下载

3.1.3 Keil

Keil µVision IDE：Keil（全称Keil µVision IDE）是一款集成开发环境（IDE），主要用于嵌入式系统的开发。它由德国Keil公司开发，现在已经被ARM公司收购，并与其MDK-ARM软件包合并成为MDK-ARM Keil软件包。

软件官网下载地址：https://www.keil.com/download/product/

图 8 Keil下载

截止目前官网的最新版本为 5.41，下载完成后一路安装即可。

3.1.4 CH340 驱动

CH340 是一种常用的USB转串口芯片，广泛应用于各种嵌入式设备中。它支持全速USB接口，能够实现USB到串行通信的转换，为开发者提供了一种便捷的串口通信解决方案。在使用CH340 驱动时，需要确保驱动程序与操作系统兼容，并正确安装，以便设备能够被计算机识别和使用。

CH340 驱动下载地址：https://www.wch.cn/downloads/CH341SER_EXE.html

图 9 CH340 下载页面

直接下载安装即可。

3.2 Stm32CubeMX工程建立

假设你现在已经安装好STM32CubeMX，双击打开软件，点击下图中框选的部分，新建一个工程：

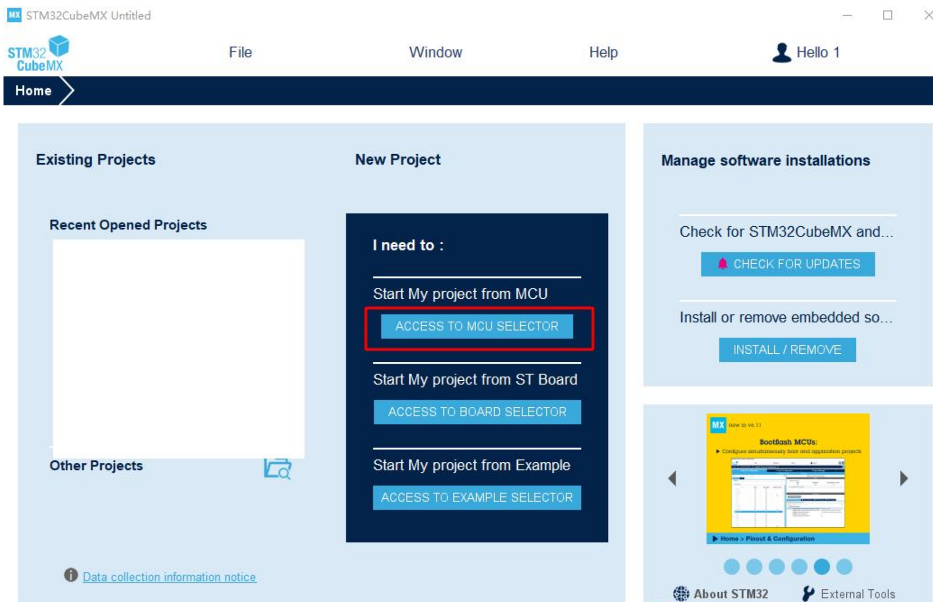

图 10 STM32CubeMX建立工程

在弹出来的页面中搜索相应的单片机型号，双击型号确认：

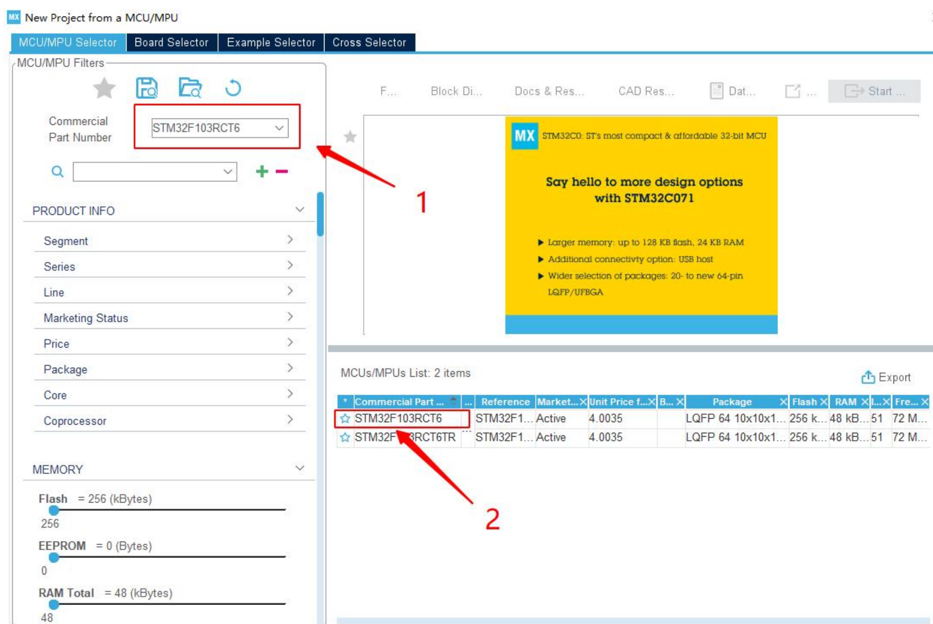

图 11 STM32CubeMX型号选择

进入到单片机的配置界面：

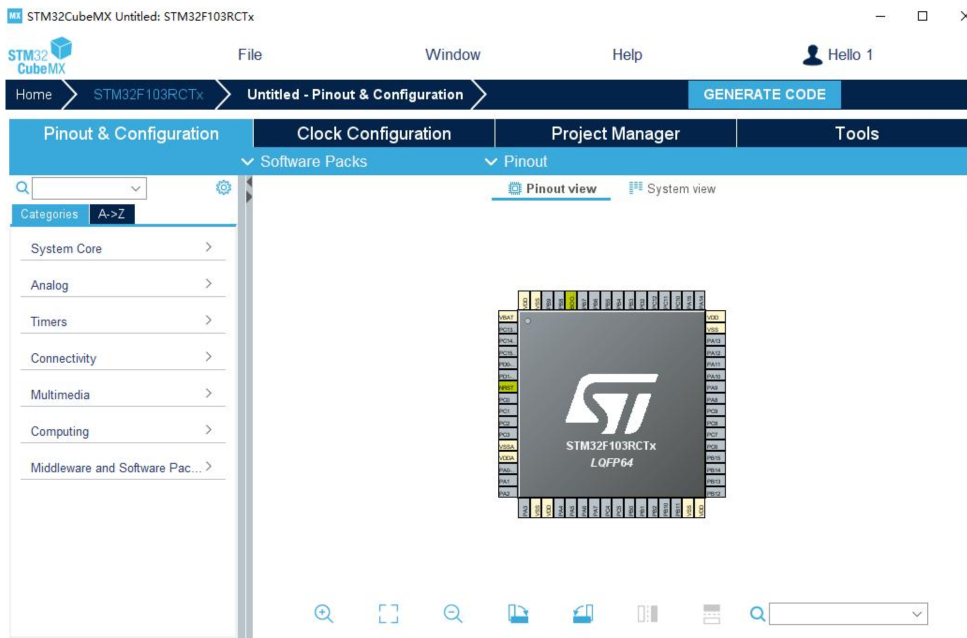

图 12 芯片配置

我们需要在单片机的配置界面进行一些简单设置，需要配置单片机的时钟和调试端口，根据自己的情况自行设定，我这里将调试端口设为SW，时钟设为64Mhz，使用内部时钟源（这里可以不用太关注时钟源的设置）。

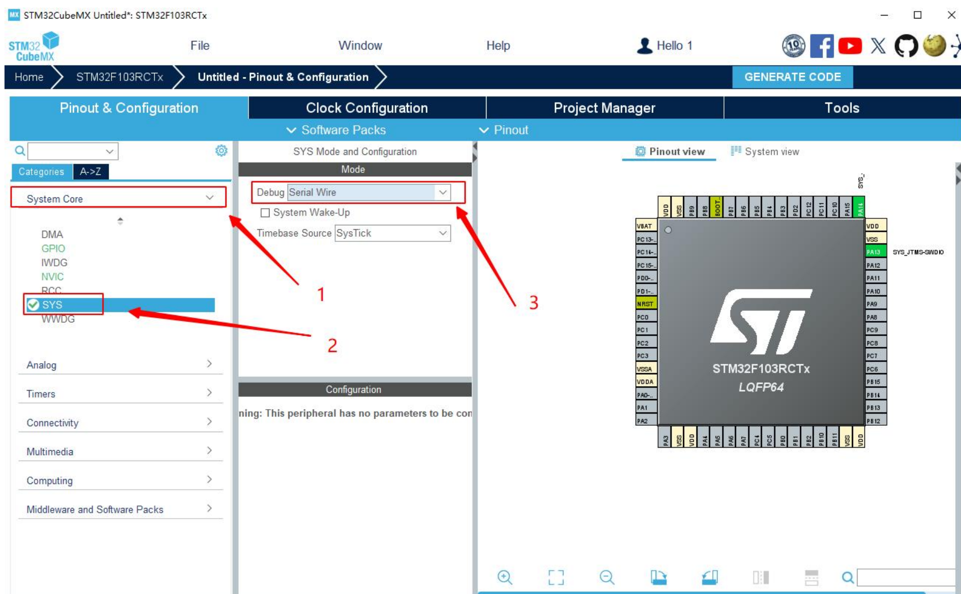

图 13 芯片SW设置

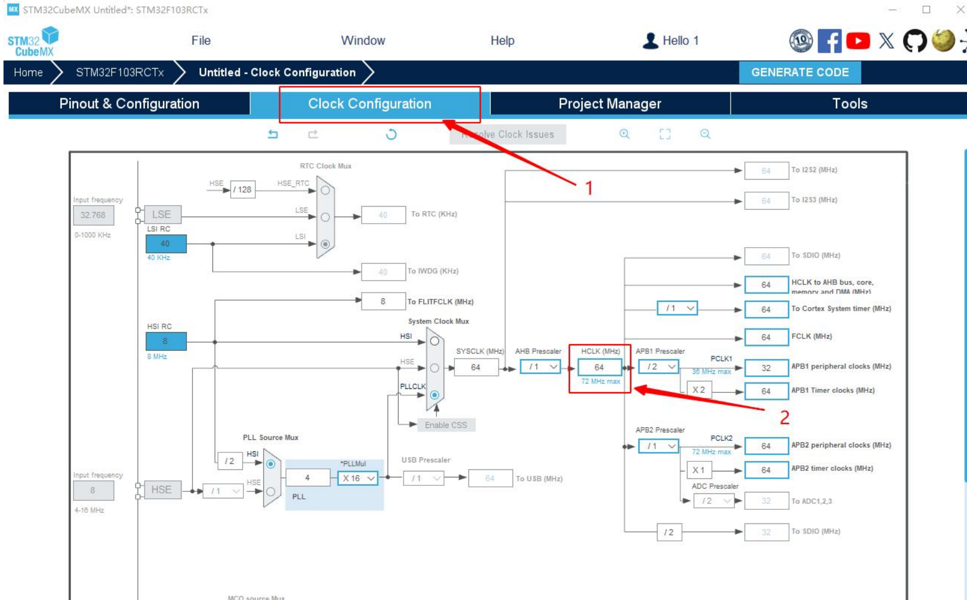

图 14 芯片时钟设置

再设置一个引脚用作LED灯闪烁，我这里的开发板上是PC7 作为LED引脚：

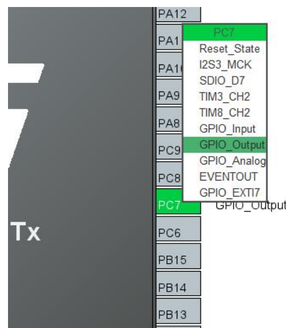

图 15 芯片LED引脚设置

接下来设置工程的名字，将工程导出为Keil工程：

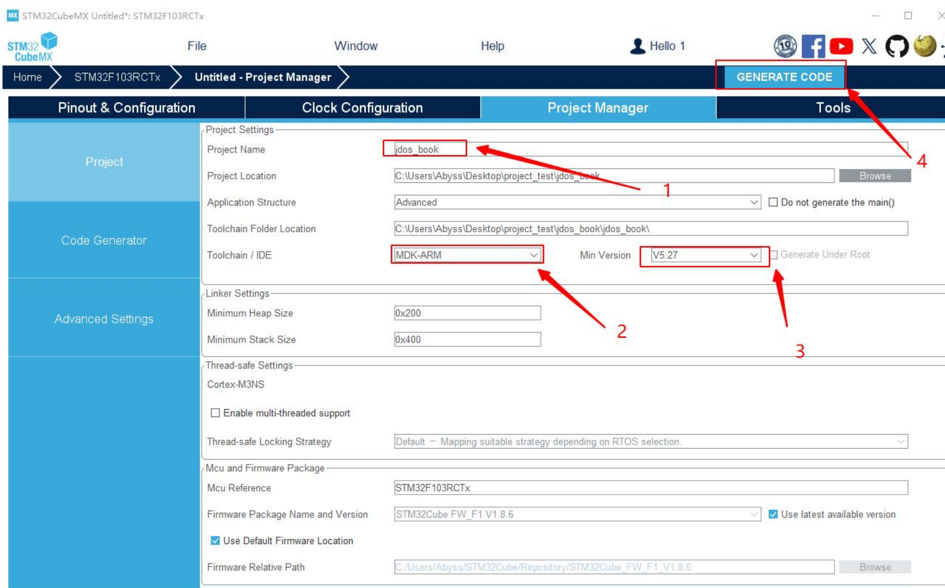

图 16 导出 Keil工程设置

完成导出后，将弹出一个确认页面，可以通过该页面直接打开Keil工程文件，或者选择直接关闭页面。

3.3 打开Keil工程并编写LED闪烁程序

用Keil打开STM32CubeMX生成的工程文件：

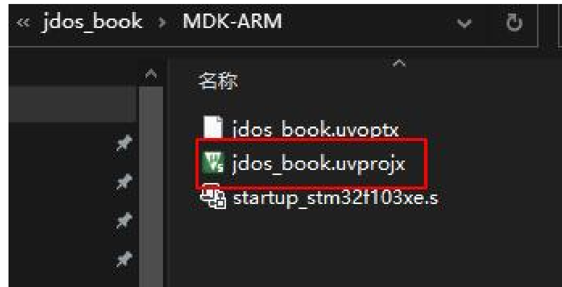

图 17 Keil工程文件

打开工程后，会注意到左侧栏中包含了一些文件，在这些文件中，启动文件负责建立中断向量表，当内核复位时，执行流程将从这个文件开始，至于main文件，它是C语言程序执行的起点，HAL库文件则包含了ST公司HAL库的实现函数，最后，系统时钟文件主要涉及系统时钟的配置。见下图：

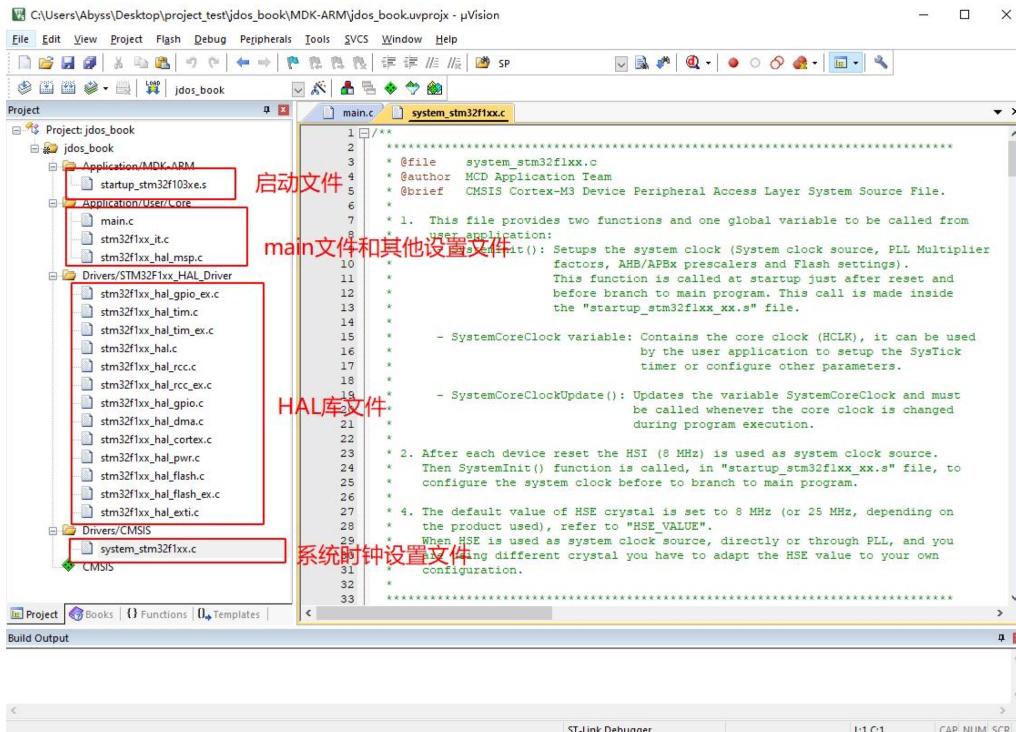

图 18 Keil工程目录

每个程序的起点往往是一个简单的“Hello, World!”示例，然而，在这里我不打算输出“Hello, World!”，这个示例放到后面实现，我们现在将着手编写一个更为基础的程序：一个简单的LED闪烁程序。

```
HAL_Delay(500); //延时500ms
HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_7); //翻转LED引脚
```

将以上代码复制到main函数的while循环中：

```
 92
 93   /* Infinite loop */
 94   /* USER CODE BEGIN WHILE */
 95   while (1)
 96   {
 97     HAL_Delay(500); //延时500ms
 98     HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_7);
 99     /* USER CODE END WHILE */
100
101     /* USER CODE BEGIN 3 */
102   }
103   /* USER CODE END 3 */
104 }
105
```

图 19 LED闪烁程序

编译代码，完成后，利用Keil下载至单片机，这里假设你的计算机已通过ST-LINK正确与单片机连接，并且相应的驱动程序已正确安装。

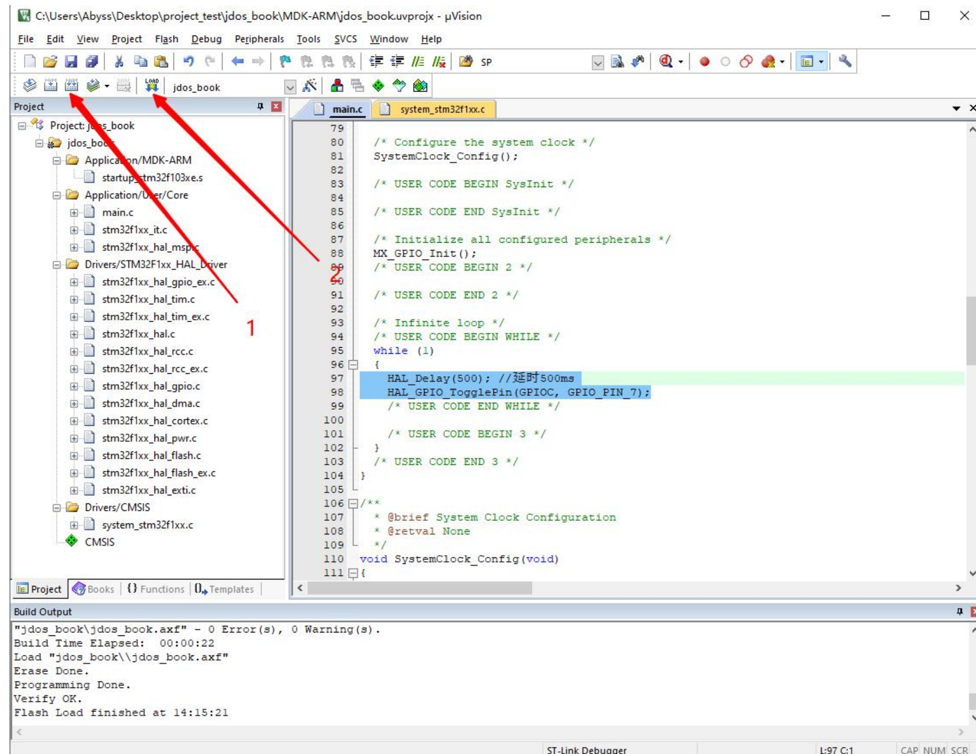

图 20 Keil编译下载

程序下载完成后，通常需要手动按下开发板上的RST复位按钮（具体操作可能因开发环境的不同而有所差异），随后将会观察到开发板上的LED指示灯开始

闪烁，这标志着工程构建已经成功完成。

小结

本章详细阐述了jdos开发环境的配置过程，包括使用STM32CubeMX工具生成Keil工程的方法，以及如何通过Keil进行代码的编译和下载。此外，本章还展示了如何编写一个简单控制LED灯闪烁的程序。

第四章 任务理论构建

在上一章节中，我们已经成功地利用STM32CubeMX创建了Keil工程。现在，让我们抑制住内心的激动，来构建一些关于jdos的基础设计理论。

4.1 任务的概念

我们将任务（Task）界定为程序执行的基本单元，每个任务都各自的上下文和状态，其中包含了程序的全部信息。任务通过优先级进行管理，确保高优先级的任务相较于低优先级的任务有更高的执行机会，每个任务都配备了独立的栈。

此外，我们需定义一个空闲任务，这是系统中优先级最低的任务。它仅在系统内无其他任务待执行时启动。空闲任务的主要目的是确保处理器在无其他工作可做时仍能保持运行状态，避免处理器进入空闲状态，当所有其他任务均无工作可执行时，空闲任务随即投入运行。

4.2 任务切换原理

任务切换的原理主要涉及两个步骤，首先，必须保存当前任务的状态信息，这涵盖了所有寄存器的状态，接着，取出下一个任务的状态信息（即下一个任务的所有寄存器状态）来覆盖当前寄存器的状态，之后继续执行。通过这种方式，CPU将开始运行下一个任务，从而实现任务的切换。

在系统运行期间，盲目地进行任务切换可能会引起系统混乱并降低效率。因此，我们需要通过定义特定事件来通知系统，以便在适当的时候进行任务切换。

（1）系统时钟事件：我们这里定义一个系统时钟，系统时钟由SysTick产生，每当系统时钟来临时，产生系统时钟事件。

（2）任务结束事件：当正在运行的任务运行完成后，此时系统还在等待系统时钟的来临，白白浪费了时间，任务结束后应产生任务结束事件。

（3）任务主动切换事件：某些任务在运行中要求主动切换下一个任务，通常是任务需要加入延时等待，此时产生任务主动切换事件。

以上事件共同构成了系统触发任务切换的条件。

4.3 任务的状态

通常情况下，CPU在任一时刻仅能处理一个任务。因此，我们需要对任务的状态进行明确的定义，以便于后续的任务管理。

运行状态：指当前任务正在CPU中执行。

就绪状态：意味着当前任务已经准备完毕，正等待系统进行调度。

暂停状态：当前任务已被搁置，处于等待用户指令以继续运行的状态。

延时状态：当前任务正处于延时阶段，一旦延时结束，它将转换为就绪状态。

以上，我们对任务的四种基本状态进行了简单的定义。

4.4 任务的堆栈

在执行任务的过程中，不可避免地会涉及到堆栈的使用，这是计算机科学中的两个基础概念。

堆（Heap）：通常由用户自行管理分配与释放，例如在标准库中，malloc函数用于分配内存，而free函数用于释放内存。通过malloc分配的内存区域即位于堆上。

栈（Stack）：由系统自动进行分配和释放，例如函数的局部变量。栈中存储了函数执行过程中的数据。

堆的内存地址是递增的，而栈的内存地址是递减的。在创建任务时，应当为其分配适量的内存空间，以用作任务的栈。

4.5 时间片轮转

时间片轮转调度是一种用于分时系统的调度策略。我们将时间片定义为系统的最小时间单位，时间片的定时由系统时钟事件负责提供。时间片通常情况下为1ms，时间片的长度对任务执行效率具有显著影响。若时间片过长，将削弱任务的实时性，反之，若时间片过短，则会导致系统频繁地进行任务切换，从而降低整体效率。每当系统时钟事件触发，即标志着一个时间片的结束，若任务在时间片内未能完成，则会被剥夺CPU的使用权，等待下一次执行，而系统会将CPU使用权转交给其他任务。为了实现时间片轮转，需要将所有就绪的任务排入队列中，

以便系统进行有效的调度。

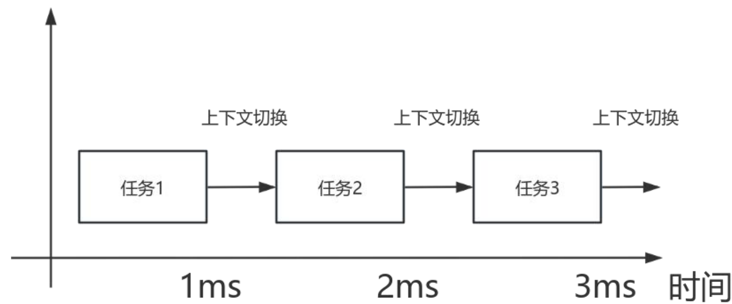

图 21 时间片轮转任务示意

4.6 任务链表设计

链表是一种常见的基础数据结构，它由一系列节点组成，每个节点包含数据部分和指向下一个节点的指针，在C语言中，链表的节点通常使用结构体来定义。链表适合管理动态数据集合，因为它可以高效地插入和删除元素。链表的每个节点包含数据部分和指向下一个节点的指针。这种结构使得在链表中添加或移除节点时，只需调整相邻节点之间的指针即可，而不需要像数组那样移动大量元素。

链表有多种类型，包括单向链表、双向链表和循环链表。单向链表的节点只包含一个指针，指向下一个节点；双向链表的节点包含两个指针，分别指向前一个节点和下一个节点，这使得双向链表在某些操作上更加高效；循环链表其实就是双向链表的最后一个节点的指针指向第一个节点，形成一个环状结构，适用于某些特定场景。

为了更高效地管理多项任务，我们采用循环链表结构，这种结构便于实现一个高效的队列系统。

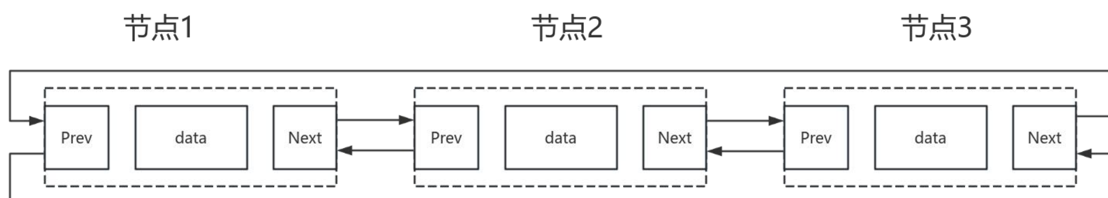

图 22 循环链表结构

小结

本章我们对jdos中的任务进行了基础设计，并构建了一些核心概念。

第五章 流程与规则

在着手编写jdos之前，我们还要对基本的流程与标准有所了解，本章将为实现jdos提供必要的理论基础。

5.1 C语言编译过程

C语言编译过程主要分为预处理、编译、汇编和链接四个过程。

1. 预处理：预处理阶段主要处理源代码文件中的预处理指令，如宏定义、文件包含、条件编译等。预处理器会根据这些指令对源代码进行相应的处理，生成预处理后的代码文件。

2. 编译：编译阶段将预处理后的代码转换成汇编代码。编译器会检查语法错误，并将高级语言指令转换为机器语言指令，但此时的指令还是以汇编语言的形式存在。

3. 汇编：汇编阶段将编译器生成的汇编代码转换成机器代码，即目标文件。汇编器会将汇编指令转换为处理器能够理解的机器指令。

4. 链接：链接阶段将一个或多个目标文件与库文件链接在一起，生成最终的可执行文件。链接器负责解决目标文件之间的符号引用，确保程序中所有引用的函数和变量都能正确地找到其定义。

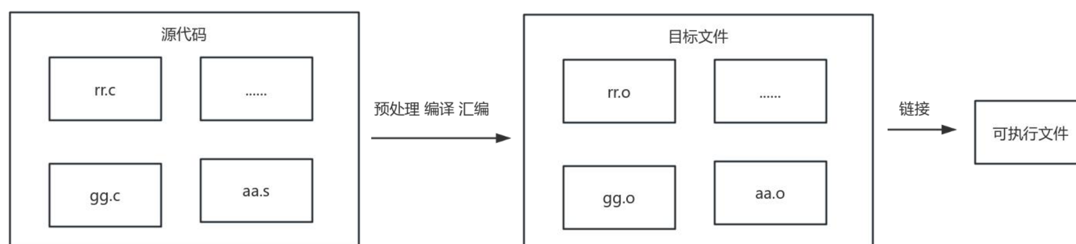

图 23 编译过程

当然，我们无需深入了解编译过程，只需掌握其大致流程即可；其余的细节，编译器已经为我们妥善处理。

在编译过程中，我们还需要和遵循一些特定的编程标准，如ATPCS（ARM-Thumb Procedure Call Standard），它规定了函数调用时的寄存器使用规

则和堆栈行为，以确保不同编译器生成的代码能够正确地交互。

此外，了解一些常用的汇编指令对于理解编译过程和后续的调试工作也是非常有帮助的。例如，了解如何使用汇编语言进行函数调用、参数传递、返回值处理等操作，可以帮助我们更好地理解程序的底层行为。

自动入栈与出栈是编译器在编译过程中自动处理的，它确保了函数调用时局部变量的存储和恢复。当函数被调用时，编译器会自动将调用者的上下文信息（如返回地址、参数等）压入堆栈；当函数返回时，编译器会自动从堆栈中恢复这些信息，并将控制权返回给调用者。

通过理解这些编译过程中的关键步骤和标准，我们可以更好地控制程序的构建过程，优化程序性能，并在出现问题时进行有效的调试。

5.2 启动流程

在单片机的启动流程中，当电源开启后，CPU会从复位向量地址开始执行代码。这个地址通常指向一个启动文件，该文件包含了中断向量表和复位处理函数。复位处理函数会初始化系统时钟、堆栈指针等基本硬件资源，并最终跳转到主函数main执行。

让我们打开启动文件startup_stm32f103xe.s，来看看究竟：

```
59  __Vectors       DCD     __initial_sp               ; Top of Stack
60                 DCD     Reset_Handler              ; Reset Handler
61                 DCD     NMI_Handler                ; NMI Handler
62                 DCD     HardFault_Handler          ; Hard Fault Handler
63                 DCD     MemManage_Handler          ; MPU Fault Handler
64                 DCD     BusFault_Handler           ; Bus Fault Handler
65                 DCD     UsageFault_Handler         ; Usage Fault Handler
66                 DCD     0                          ; Reserved
67                 DCD     0                          ; Reserved
68                 DCD     0                          ; Reserved
69                 DCD     0                          ; Reserved
70                 DCD     SVC_Handler                ; SVCall Handler
71                 DCD     DebugMon_Handler           ; Debug Monitor Handler
72                 DCD     0                          ; Reserved
73                 DCD     PendSV_Handler             ; PendSV Handler
74                 DCD     SysTick_Handler            ; SysTick Handler
```

图 24 中断向量表

在启动文件中，已经预先配置好了中断向量表，这是一个用于存储中断处理程序地址的数据结构。这种预设极大地节省了我们的时间和精力，因为无需手动设置这些中断处理程序的地址。仔细观察代码的第 59 行，我们可以发现这是单片机在复位后执行的第一条指令。这条指令的作用是初始化栈指针，为后续的程

序运行提供必要的支持。紧接着这条指令之后，系统将执行Reset_Handler函数，也就是复位异常处理程序。这意味着，在初始化栈指针之后，系统将立即执行Reset_Handler函数，以确保系统在复位后能够正确地进行初始化操作。

在后续的代码中，我们还注意到SVC_Handler、PendSV_Handler和SysTick_Handler等异常处理程序。这些异常处理程序已经在之前的介绍中有所提及，它们分别对应着系统调用异常、可悬起的系统服务调用异常和系统定时器异常。这些异常处理程序已经在中断向量表中得到了明确的定义和配置，因此在系统运行过程中，一旦发生这些异常，系统将能够迅速找到对应的处理程序，进行相应的异常处理操作。

接下来，我们将继续查看Reset_Handler异常处理函数：

```
144  Reset_Handler    PROC
145                   EXPORT  Reset_Handler             [WEAK]
146                   IMPORT  __main
147                   IMPORT  SystemInit
148                   LDR     R0, =SystemInit
149                   BLX     R0
150                   LDR     R0, =__main
151                   BX      R0
152                   ENDP
```

图 25 Reset_Handler处理函数

Reset_Handler PROC：定义一个名为Reset_Handler的函数，PROC表示这是一段函数代码。

EXPORT Reset_Handler [WEAK]：将Reset_Handler标记为一个弱引用的导出符号，如果其他地方没有定义Reset_Handler，链接器将使用这个定义；如果其他地方有定义，链接器将忽略这个定义。

IMPORT __main：导入名为__main的符号，这是C语言中主函数的入口点。

IMPORT SystemInit：导入名为SystemInit的符号，这是用于系统初始化的函数，包括时钟初始化等。

LDR R0, =SystemInit：将SystemInit函数的地址加载到寄存器R0 中。

BLX R0：使用R0 寄存器中的地址来调用SystemInit函数。BLX指令用于调用函数，并且它还会将返回地址保存到链接寄存器（LR），以便函数执行完毕后能够返回。

LDR R0, =__main：将__main函数的地址加载到寄存器R0 中。

BX R0：使用R0 寄存器中的地址跳转到__main函数，开始执行main程序。

ENDP：标记Reset_Handler函数的结束。

通过执行这段程序，单片机成功地完成了初始设置，并顺利地跳转至主函数main。在这个过程中，我们终于理解了为什么C语言的标准规定程序的入口点必须是main函数。现在，我们完全可以通过修改main函数来实现跳转至我们自定义的函数，但这种做法在大多数情况下其实并没有太大的实际意义。毕竟，main函数本质上只是作为程序的入口点，它仅仅代表了一个标识符，一个指向C语言程序执行起始位置的标识符。

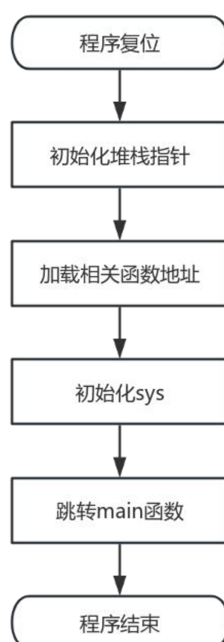

图 26 启动流程图

5.3 ATPCS标准

在C语言和汇编语言之间进行交互时，为了确保两者能够正确地互相调用，必须遵循一定的标准和规范。这些交互主要遵循的是ARM过程调用标准（ARM Procedure Call Standard），也被称为ARM Transfer Procedure Call Standard (ATPCS)。在探讨ATPCS时，我们重点关注其参数传递规则。

在ARM架构下，寄存器R0 至R3 被指定用于传递函数参数。在ATPCS中，这些寄存器分别被称为a1 至a4。当调用子程序时，这些寄存器的内容通常无需

保存和恢复，因为它们专门用于传递参数和返回值。这意味着，当C语言调用一个汇编程序时，可以一次简单地传递四个参数，这些参数的数据依次存放在R0至R3 中。相应地，当汇编函数需要调用C函数并传递参数时，只需将数据存入R0 至R3 即可。在此，我们不探讨传递超过四个参数的情况。

参数传递规则确保了在ARM架构上，无论是用C语言编写的程序还是用汇编语言编写的程序，都能够遵循相同的参数传递约定，从而实现相互调用。这种一致性使得混合编程变得更加容易和可靠。

ATPCS不仅仅包括参数传递规则，它还涵盖了其他许多规则，例如堆栈的使用、寄存器的保存和恢复等。为了更全面地了解ATPCS，建议查阅相关的官方文档，以便获得更详细的信息和深入的理解。

5.4 常用的汇编指令

Thumb-2 指令集是 16 位Thumb指令集的扩展版本，它不仅兼容原有的 16 位指令，还引入了 32 位指令，使得这两种长度的指令可以在同一个系统中共存。这种设计显著提升了代码的密度，使得程序更加紧凑，同时也提高了执行效率，因为 32 位指令可以执行更复杂的操作。Cortex-M3 处理器是专为实时应用设计的一款高性能处理器，它仅支持Thumb-2 指令集，这种设计选择消除了在 16 位和 32 位指令之间切换状态的开销。这种单一指令集的支持使得Cortex-M3 处理器能够更高效地执行指令，从而提高整体性能。此外，Thumb-2 指令集还引入了一些新的指令集特性，例如位段操作和除法指令，这些新特性进一步增强了处理器的功能，使得开发者能够更灵活地编写高效且复杂的程序代码。

我们不深入每一条指令与它的特性，这里仅对一些常用的指令进行解释，更多指令详情请参阅《DDI0403E_e_armv7m_arm》。

表 4 常用的汇编指令

指令	功能
ADD	加法
AND	按位与
CPM	比较两个数并且更新标志
MOV	寄存器加载数据，既能用于寄存器间的传输，也能用于加载立即数

ORR	按位或
B	跳转
BL	跳转并保存返回地址到LR中
CBZ	比较，如果结果为 0 就跳转
CBNZ	比较，如果结果非 0 就跳转
LDR	从存储器中加载字到一个寄存器中
LDRB	从存储器中加载字节到一个寄存器中
STR	把一个寄存器按字存储到存储器中
PUSH	可以将多个寄存器压入到栈中
POP	可以从栈中弹出多个值到寄存器中
SVC	系统服务调用
NOP	无操作
CPSIE	使能PRIMASK
CPSID	除能PRIMASK
MRS	加载特殊功能寄存器的值到通用寄存器
MSR	存储通用寄存器的值到特殊功能寄存器

大多数指令都会带有特定的后缀，这些后缀用于表示指令的不同功能和操作。常见的后缀包括EQ、NE、LT和GT等，这些后缀分别对应着不同的状态和条件。具体来说，EQ通常表示“等于”（Equal），用于判断两个值是否相等；NE表示“不等于”（Not Equal），用于判断两个值是否不相等；LT表示“小于”（Less Than），用于判断一个值是否小于另一个值；而GT表示“大于”（Greater Than），用于判断一个值是否大于另一个值。这些后缀在条件分支、循环控制以及逻辑判断等场景中非常有用，能够编写更加高效和精确的代码。通过这些后缀的使用，可以实现复杂的逻辑判断和数据处理，从而提高程序的灵活性和功能性。

5.5 自动入栈与出栈

当Cortex-M3 处理器在执行任务的过程中，会遇到了一些异常情况并触发了中断响应时，处理器会自动执行一系列的入栈操作。这些操作的主要目的是为了保护当前的运行环境，确保在异常处理完毕之后，系统能够顺利地恢复到中断发

生之前的状态。在中断服务程序执行完毕，并且准备返回到正常的执行流程时，处理器会自动进行出栈操作。这一过程确保了现场的保护与恢复，使得系统能够在处理完异常后，继续以稳定和可靠的方式运行。

在自动入栈的过程中，处理器会按照一个特定的顺序来保存寄存器的值。这些寄存器包括xPSR、R15（也就是程序计数器PC）、R14（链接寄存器LR），以及通用寄存器R3、R2、R1 和R0。这一顺序的设定，确保了在中断处理过程中，所有关键的寄存器状态都被安全地保存在栈中，从而避免了数据丢失或状态混乱的问题。

相应地，在自动出栈操作中，处理器会按照与入栈时相反的顺序，将栈中的数据依次弹出。这些数据会被依次恢复到对应的寄存器中，从而恢复中断发生前的寄存器状态。通过这种方式，处理器能够在中断服务程序执行完毕后，准确地恢复到中断前的运行状态，继续执行后续的指令。这一机制对于确保系统的稳定性和可靠性至关重要，特别是在实时系统和嵌入式应用中，中断处理的及时性和准确性直接关系到系统的整体性能和可靠性。因此，Cortex-M3 处理器通过这种自动的入栈和出栈机制，有效地保障了系统在面对异常情况时的鲁棒性和响应能力。

小结

本章详尽阐述了Keil工程启动文件的启动流程，简单的介绍了ATPCS标准、一系列常用的汇编指令以及自动出入栈的规则。

第六章 内核编写

经过前面几章的铺垫，现在，我们可以开始着手实现jdos中最为关键的部分——jdos内核代码编写。

6.1 头文件定义

打开我们前面创建好的Keil工程，创建 3 个新的文件，分别命名为jdos.h、jdos.c和jdos.s：

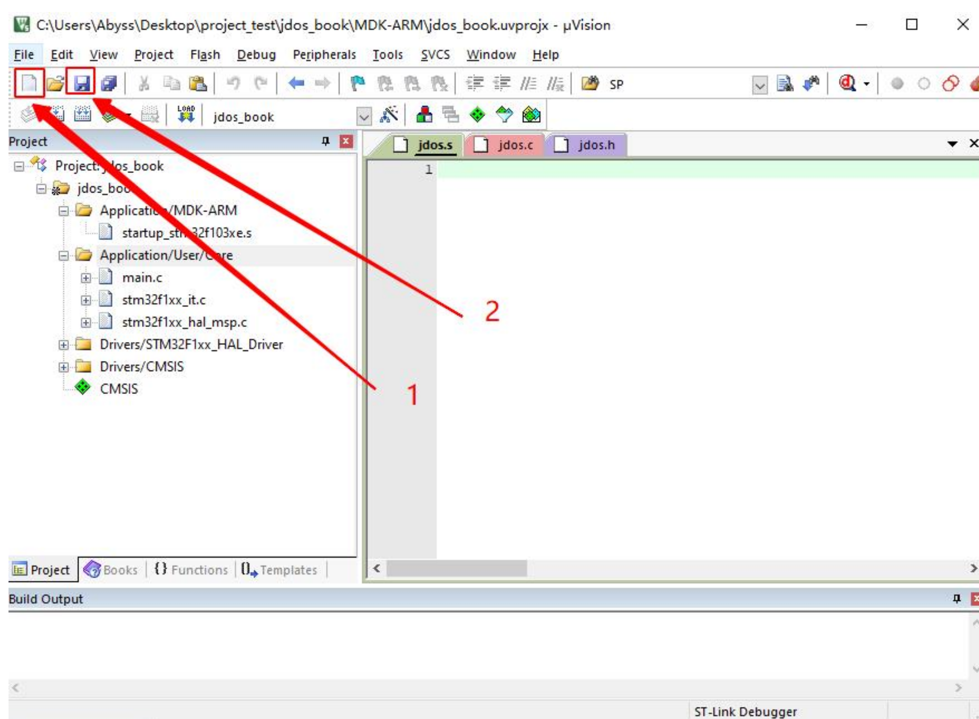

图 27 Keil新建文件

jdos.h：为jdos的头文件，里面包括所有的定义。

jdos.c：为jdos的c文件，jdos核心功能实现。

jdos.s：为jdos必要的汇编文件。

将这 3 个文件添加到工程中：

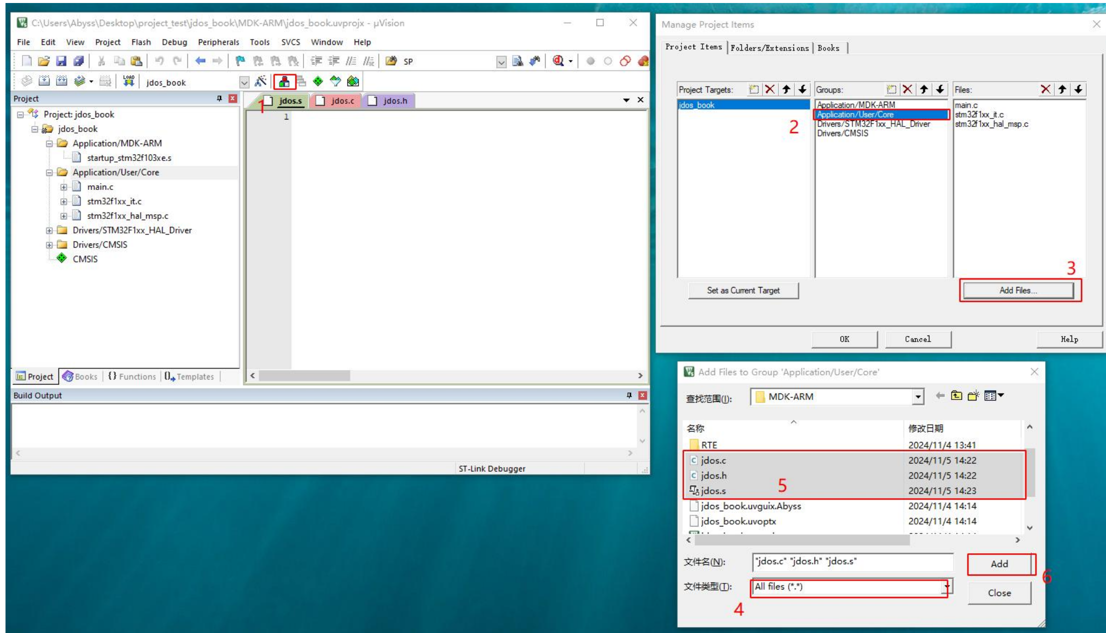

图 28 Keil添加文件

将jdos.h的头文件路径添加到工程中：

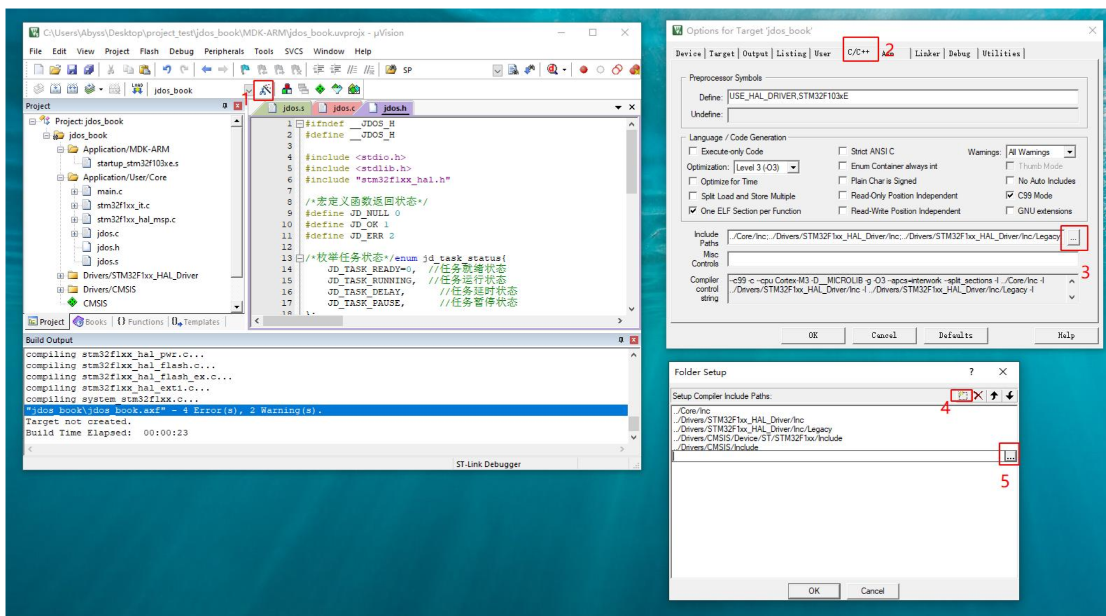

图 29 添加头文件

首先，在jdos.h头文件中进行一些基础的定义工作，通过宏定义来规范函数的通用返回值，设定默认的堆栈大小为 512 字节，并对系统时钟进行相应的定义。

```
#ifndef __JDOS_H
#define __JDOS_H
/*宏定义函数返回状态*/
#define JD_NULL 0
#define JD_OK 1
#define JD_ERR 2
```

```
/*系统默认堆栈大小*/
#define JD_DEFAULT_STACK_SIZE 512
/*系统时钟，单位ms*/
unsigned long jd_time = 0;
#endif
```

在先前章节中，我们提到了任务具有四种状态，现在我们对这四种状态进行枚举。

```
/*枚举任务状态*/
enum jd_task_status{
JD_TASK_READY=0,         //任务就绪状态
JD_TASK_RUNNING,         //任务运行状态
JD_TASK_DELAY,               //任务延时状态
JD_TASK_PAUSE,               //任务暂停状态
};
```

在先前章节中，我们已经讨论了Cortex-M3 的寄存器。现在，我们通过结构体的方式对这些寄存器进行定义，以便于后续在分配任务栈空间时能够便捷地保存任务上下文。值得注意的是，这些寄存器在结构体中的排列遵循着特定的顺序。

```
/*定义所有寄存器，根据入栈规则有先后顺序*/
struct all_register
{
//手动入栈
unsigned long r4;
unsigned long r5;
unsigned long r6;
unsigned long r7;
unsigned long r8;
unsigned long r9;
unsigned long r10;
unsigned long r11;
//自动入栈
unsigned long r0;
unsigned long r1;
unsigned long r2;
unsigned long r3;
unsigned long r12;
unsigned long lr;
unsigned long pc;
unsigned long xpsr;
};
```

接下来我们需要定义任务控制块，它是一个包含了任务的所有相关信息的结构体，通过维护任务控制块，我们能够有效地控制和管理任务的执行，我们在任

务控制块中定义了双向链表的结构，任务入口函数，当前任务状态，堆栈大小，堆栈指针，堆栈地址以及延时溢出时间。

```
/*定义任务节点*/
struct jd_task{
struct jd_task *previous;         //指向上一个节点
void (*entry)();                  //指向任务入口函数
enum jd_task_status status;       //当前任务状态
unsigned long stack_size;         //堆栈大小
unsigned long stack_sp; //堆栈指针
unsigned long stack_origin_addr; //堆栈起始地址
unsigned long timeout; //延时溢出时间，单位ms，为0 则没有延时
struct jd_task *next;    //指向下一个节点
};
```

定义好任务控制块的数据结构后，接下来需要实现相关的操作函数，如创建任务、删除任务、暂停任务等。

6.2 任务创建

每个任务都配备了一个独立的栈，实质上，这意味着将任务的所有数据存储在这个栈上，栈空间的申请涉及到内存管理，但目前我们面临的问题是创建任务，而非内存管理，因此，我们暂时使用标准库提供的分配函数，也就是malloc和free。

在jdos.h头文件中引入相关库：

```
#include <stdio.h>
#include <stdlib.h>
#include "stm32f1xx_hal.h"
```

记得在Keil工程中勾选Use MicroLIB，否则标准库的某些函数使用上会有问题:

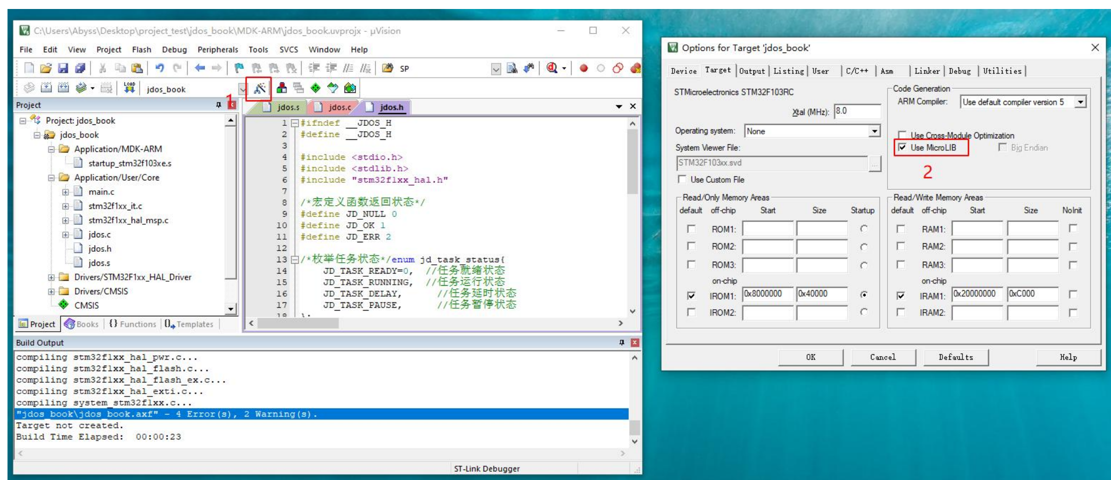

图 30 勾选库

接下来在jdos.c中编写代码，定义所需的全局变量，以追踪第一个任务的指针，创建一个全局任务链表指针，以及当前任务和下一个任务的栈指针。

```
#include "jdos.h"
struct jd_task *jd_task_sp_frist= NULL;    //用于保存节点第一个任务位置
struct jd_task *jd_task_sp = NULL;         //创建一个任务节点指针
unsigned long *jd_task_stack_sp = NULL;           //创建当前任务堆栈指针的地址
unsigned long *jd_task_next_stack_sp = NULL;      //创建下一个任务堆栈指针的地址
```

编写栈空间申请函数：

```
/*申请任务空间
* jd_task_sp: 节点指针
* stack_size: 堆栈大小
* return: JD_OK或JD_ERR
*/struct jd_task * jd_request_space(unsigned int stack_size)
{
struct jd_task *jd_task; //（1）
jd_task = (struct jd_task *)malloc(sizeof(struct jd_task)); //分配空间（2）
if(jd_task==NULL)return JD_NULL; //判断分配空间是否成功
jd_task->stack_sp = (unsigned long)malloc(stack_size); //申请堆栈空间（3）
if(jd_task->stack_sp==NULL)return JD_NULL; //判断分配空间是否成功
jd_task->stack_origin_addr = jd_task->stack_sp; //记录栈顶指针 （4）
return jd_task;//（5）
}
```

在上述代码中（1）定义了一个任务控制块的指针；

（2）使用标准库中的malloc函数申请一个任务控制块的空间；

（3）使用标准库中的malloc函数申请stack_size大小的空间用作任务栈空间；

（4）记录这个任务栈的栈顶指针；

（5）返回任务控制块的指针。

申请空间后，将任务的信息填入空间，也就是创建任务：

```
/*创建任务
* task_entry:函数入口
* stack_size: 任务栈大小
* return: 返回当前任务节点指针
*/struct jd_task *jd_task_create(void (*task_entry)(),unsigned int stack_size)
{
struct jd_task *jd_new_task = NULL;        //创建一个任务节点指针（1）
jd_new_task = jd_request_space(JD_DEFAULT_STACK_SIZE);       //（2）
if(jd_new_task==JD_NULL)return JD_NULL;        //申请空间
if(jd_task_sp!=NULL)
```

```
    {
    jd_new_task->previous = jd_task_sp; //新节点指向当前节点
    jd_new_task->next = jd_task_sp->next;//新节点指向下一个节点
    jd_task_sp->next->previous = jd_new_task; //下一个节点指向当前节点
    jd_task_sp->next = jd_new_task; //当前节点指向新节点
    } //（3）
    jd_new_task->timeout = 0; //没有延时时间（4）
    jd_new_task->entry = task_entry; //任务入口（5）
    jd_new_task->status = JD_TASK_PAUSE; //创建任务，状态为暂停状态，等待启动（6）
    jd_new_task->stack_size = stack_size; //记录当前任务堆栈大小（7）
    jd_new_task->stack_sp =
(jd_new_task->stack_origin_addr+JD_DEFAULT_STACK_SIZE-sizeof(struct
all_register))&0xfffffffc; //腾出寄存器的空间（8）
    struct all_register *stack_register = (struct all_register *)jd_new_task->stack_sp;
//将指针转换成寄存器指针（9）
    //将任务运行数据搬移到内存中
    stack_register->r0 = 0;
    stack_register->r1 = 0;
    stack_register->r2 = 0;
    stack_register->r3 = 0;
    stack_register->r12 = 0;//（10）
    stack_register->lr = (unsigned long)jd_new_task->entry;//（11）
    stack_register->pc = (unsigned long)jd_new_task->entry;//（12）
    stack_register->xpsr = 0x01000000L; //由于Armv7-M只支持执行Thumb指令，因此必须始终将
其值保持为1（13）
    return jd_new_task; //返回当前任务节点//（14）
    }
```

上述代码中（1）定义了一个任务控制块指针；

（2）申请一个JD_DEFAULT_STACK_SIZE大小的任务栈空间，并把返回任务控制块指针；

（3）将当前任务加入全局任务链表；

（4）设置当前默认延长时间为0，即没有延时；

（5）记录任务程序的入口函数；

（6）设置任务的默认状态为暂停状态，等待用户启动；

（7）记录当前申请的任务栈大小；

（8）任务栈空间地址从stack_origin_addr开始，栈的空间大小设定为JD_DEFAULT_STACK_SIZE，那么栈顶的实际地址应为

JD_DEFAULT_STACK_SIZE减去1字节。在任务栈顶，我们预留了用于all_register寄存器的空间，并更新了任务栈指针stack_sp。由于地址增长方向向上，而栈的增长方向向下，因此任务栈指针stack_sp实际上直接指向该all_register寄存器空间的起始地址。代码中执行的&0xfffffffc操作是为了确保内存对齐，如果申请的空间不是对齐的，将导致栈指针地址不是对齐的。在ARM架构中，内存对齐可以简单理解为地址是否能被某个数整除，我们常说的4字节对齐（4字节也就是32位，Cortex-M3是32位的，一般要求是4字节对齐），意味着地址能被4整除。如果申请的空间不是对齐的，&0xfffffffc操作强制将低两位置0，从而得出一个4字节对齐的地址，而向着地址增长方向的额外的一点不对齐的内存将不再被使用。因此，在申请内存空间时，通常会申请4的倍数的内存空间大小，避免产生不对齐的内存，这也是为什么我们默认设置栈大小为512字节的原因（请查阅相关资料了解更多信息，内存不对齐对性能的影响是巨大的，在Cortex-M3中内存不对齐甚至无法正常运行）。

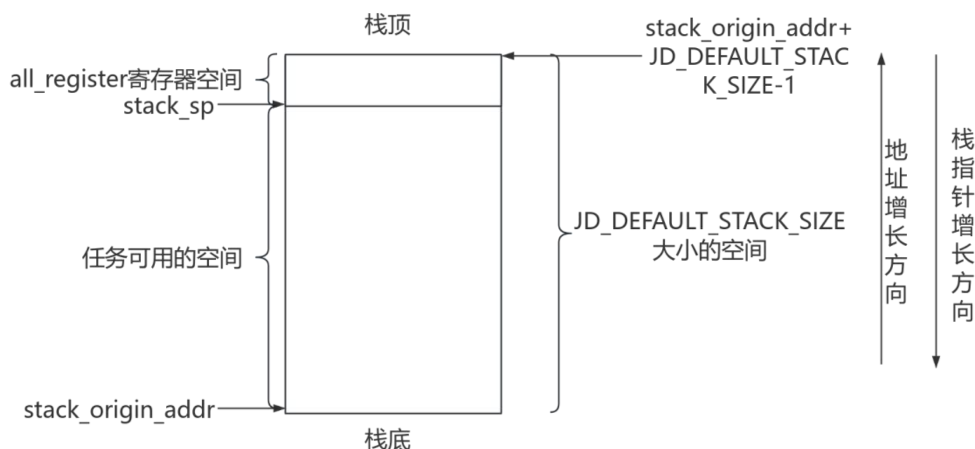

图31 任务栈空间使用示意图

（9）将任务栈指针stack_sp转换为all_register的指针，stack_register所指向的空间实际上就是all_register寄存器空间；

（10）向R0-R3和R12中写入默认数据；

（11）向链接寄存器中写入任务程序的入口函数；

（12）向程序计数器写入任务程序的入口函数；

（13）向xPSR状态寄存器写入0x01000000L，由于Armv7-M只支持执行Thumb指令，因此必须将其第24位保持为1，否则将导致失效状态（请参阅

《DDI0403E_e_armv7m_arm》）；

（14）返回任务块的指针。

我们成功编写了任务创建函数，并为任务分配了栈空间，任务创建完成后的状态为暂停状态。

6.3 任务切换

任务的上下文切换依赖于底层汇编代码，打开jdos.s文件，宏定义一些寄存器地址，并导入jdos.c文件中定义的部分变量。

```
JD_ICRS                 EQU 0XE000ED04     ;中断控制及状态寄存器
JD_PRI_14               EQU 0XE000ED23     ;PendSV的优先级设置寄存器
JD_SYSTICK_CTRL         EQU 0xE000E010     ;SysTick控制及状态寄存器
IMPORT jd_task_stack_sp
IMPORT jd_task_next_stack_sp
AREA |.text|, CODE, READONLY, ALIGN=3
```

第一次进入任务，主要目的是定位堆栈，进入程序入口，其他数据无用。

```
jd_asm_task_first_switch          PROC
                                  EXPORT  jd_asm_task_first_switch
                                  ;设置PendSV的优先级为 255
                                  LDR R3,=JD_PRI_14 ;(1)
                                  LDR R2,=0X000000FF;(2)
                                  STR R2,[R3];(3)
                                  ;第一次进入任务
                                  LDR R0,[R0];(4)
                                  MOV SP,R0;(5)
                                  MOV LR,R1;(6)
                                  CPSIE i ;开中断 （7）
                                  BX LR ;（8）
                                  ENDP
```

以上代码（1）将PendSV的优先级设置寄存器地址加载到R3；

（2）向R2 写入 0x000000FF；

（3）将R2 中的内容写入R3 的地址中，（1）（2）（3）步骤的目的是为了设置PendSV的优先级为 255，最低优先级，我们保证PendSV不会打断其他任何的中断；

（4）将R0 中地址的数据加载到R0 中，先前章节提到ATPCS标准，这里相当于外部调用传递的第一个参数，这里我们将R0 这个参数定义为任务的堆栈指

针；

（5）将外部传递进来的堆栈指针加载到堆栈指针SP中；

（6）这里我们将R1 这个参数定义为第一个任务的程序入口，将这个入口记载到链接寄存器LR中；

（7）使能系统响应所有的中断；

（8）进入任务的程序入口。

在先前章节提到了PendSV异常的使用场景，我们用汇编来实现PendSV异常的使用，这里我们先触发一下PendSV的异常。

```
jd_asm_pendsv_putup        PROC
                           EXPORT jd_asm_pendsv_putup
                           LDR R0,=JD_ICRS ;(1)
                           LDR R1,=0X10000000 ;(2)
                           STR R1,[R0] ;(3)
                           BX LR ;(4)
                           ENDP
```

在上述代码中（1）加载中断控制及状态寄存器的地址到R0 中；

（2）向R1 中写入 0x10000000；

（3）将 0x10000000 加载到中断控制及状态寄存器中，详情参考中断控制及状态寄存器的具体定义，这里目的是触发PendSV异常。

（4）程序返回。

在PendSV异常处理程序中完成任务的上下文切换。

```
jd_asm_pendsv_handler      PROC
                           EXPORT  jd_asm_pendsv_handler
                           CPSID i ;关中断（1）
                           MOV R0,SP ;（2）
                           STMFD R0!,{R4-R11};（3）
                           ;保护现场，将堆栈指针传出
                           LDR R1,=jd_task_stack_sp;（4）
                           LDR R1,[R1];（5）
                           STR R0,[R1];（6）
                           ;取下一个任务的堆栈指针,恢复现场
                           LDR R1,=jd_task_next_stack_sp;（7）
                           LDR R1,[R1];（8）
                           LDR R0,[R1];（9）
                           LDMFD R0!,{R4-R11};（10）
                           MOV SP,R0;（11）
                           CPSIE i ;开中断 （12）
```

```
BX LR; （13）
ENDP
```

在上述代码中（1）不响应所有中断（硬件故障异常除外，以下不再特指），确保在上下文切换过程中不被其他异常中断；

（2）将当前堆栈指针SP加载到R0 中，在先前章节提到过，当进入异常处理程序时，Cortex-M3 会自动将关键寄存器入栈并更新堆栈指针，在前面的代码中设置我们已将PendSV的优先级设置为了最低，并且由于PendSV的特性不会打断正在处理的中断程序，所以当进入到这段PendSV异常处理程序时，栈里面的数据一定是上一个正在运行的任务的数据，在这里我们需要把栈指针取出来，以便后面保护其他寄存器；

（3）将寄存器R4-R11 中的数据依次加载到R0 表示的地址中，也就是将R4-R11 的数据进行入栈，同时也会更新R0 中表示的地址，这里的R0 中表示的地址其实就是上一个任务的堆栈指针；

（4）（5）加载上一个任务的任务控制块中的堆栈指针地址；

（6）将更新后的堆栈指针信息保存到上一个任务的任务控制块中；

（7）（8）（9）加载下一个任务的任务控制块中的堆栈指针信息到R0

（10）将下一个任务栈中的数据出栈给R4-R11，并且更新R0 中的指针信息。

（11）将更新后的指针加载到SP堆栈指针寄存器中。

（12）打开响应中断。

（13）PendSV异常返回，此前，在（11）步中，我们已经改变了SP堆栈指针的值，为接下来的任务切换做好了准备，当PendSV异常返回时，它将自动执行一个关键步骤：将下一个任务自动入栈的寄存器数据自动出栈，确保了任务切换的顺利完成。

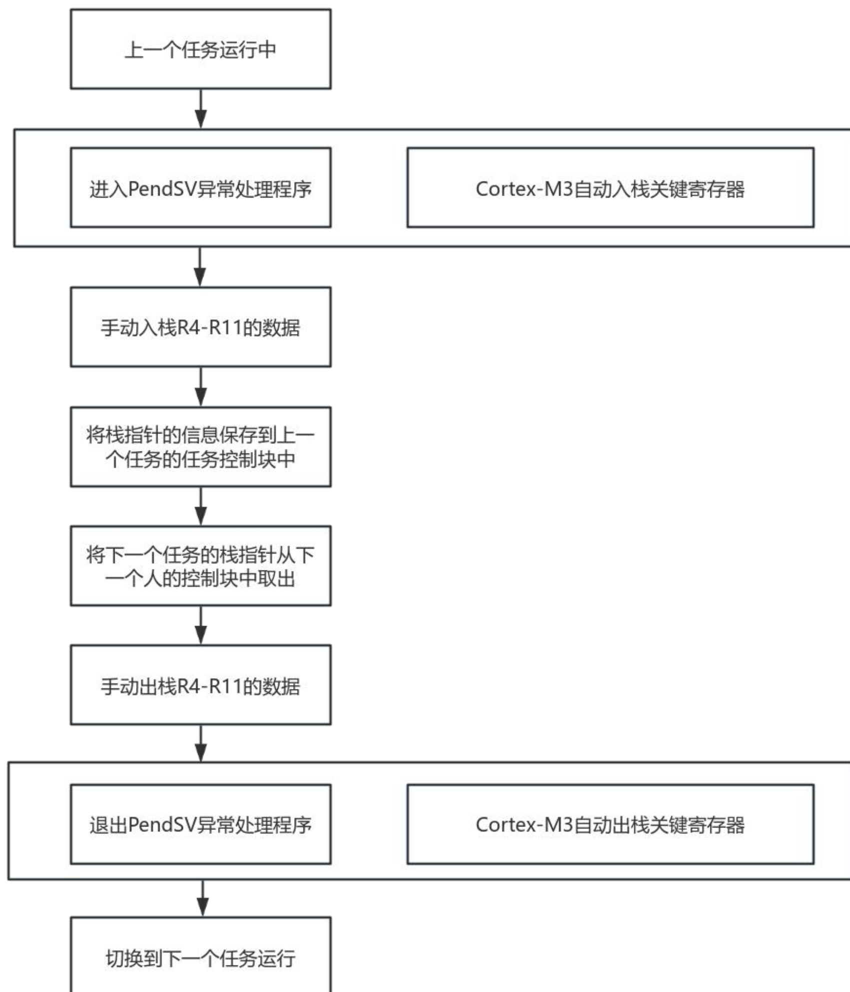

图 32 PendSV异常处理程序

我们还需要把我们刚刚编写的PendSV异常处理程序与中断向量表中的中断程序地址进行绑定，我们打开stm32f1xx_it.c文件，这里面是系统异常中断程序定义的地方，在启动文件的中断向量表中对应的中断程序地址就是定义在这里，我们找到PendSV_Handler中断处理程序，在其中添加上面编写的函数。

```
  jd_asm_pendsv_handler(); //切换上下文
```

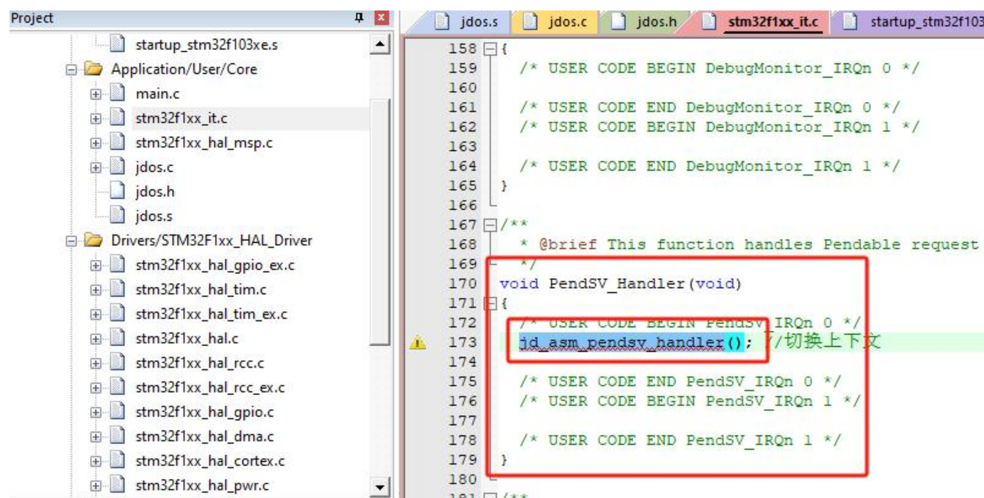

图 33 PendSV异常处理程序绑定

在jdos.s文件中，我们将开关中断的汇编指令的用法暴露给C语言，方便后续的调用。

```
jd_asm_cps_disable      PROC      ;除能在 NMI 和硬 fault 之外的所有异常
                        EXPORT jd_asm_cps_disable
                        CPSID i ;关中断
                        BX LR;
                        ENDP
jd_asm_cps_enable       PROC      ;使能中断
                        EXPORT jd_asm_cps_enable
                        CPSIE i ;开中断
                        BX LR;
                        ENDP
```

我们在jdos.c文件中准备上下文切换的必要数据，最后通过触发PendSV异常来进行上下文切换。

```
/*当前任务切换为下一个任务*/
void jd_task_switch(void)
{
    static struct jd_task *jd_task_temp;//（1）
    jd_task_temp = jd_task_sp;//（2）
    //遍历任务，选择就绪的任务 注意，如果没有任务运行，此while将陷入死循环
    while (1)
    {
        jd_task_temp = jd_task_temp->next;//（3）
        if(jd_task_temp->status==JD_TASK_READY)break;//（4）
    }
    jd_asm_cps_disable(); //关闭中断（5）
    jd_task_stack_sp = &jd_task_sp->stack_sp; //更新当前任务全局堆栈指针变量（6）
    jd_task_sp = jd_task_temp; //移动节点（7）
```

```
        jd_task_next_stack_sp = &jd_task_sp->stack_sp; //更新下一个任务全局堆栈指针变量 （8）
        jd_asm_cps_enable(); // （9）
        jd_asm_pendsv_putup(); //挂起PendSV异常 （10）
    }
```

上上述代码（1）定义一个临时的任务控制块的指针；

（2）将全局任务控制块指针加载到临时任务控制块的指针；

（3）（4）通过while来遍历链表结构，查询已经就绪的任务；

（6）关闭所有中断响应；

（6）将正在运行任务堆栈指针放在全局堆栈指针；

（7）将正在运行的任务块切换查找到的就绪任务块；

（8）将查找到的就绪任务堆栈指针放在全局下一个堆栈指针；

（9）打开中断响应；

（10）触发PendSV异常。

任务切换逻辑已经实现，在上述代码中我们可以在任务中通过手动调用jd_task_switch函数进行上下文切换，但是，先前章节提到，我们不能随意地进行任务切换，并且在之前已经明确了三个触发任务切换的条件：系统时钟事件、任务结束事件以及任务主动切换事件。现在，我们将首先着手实现其中一个条件——系统时钟事件。

6.4 系统时钟事件触发

系统时钟事件依赖SysTick系统滴答定时器触发，这里在jdos.s文件中添加初始化SysTick初始化代码，以下代码设置重装载的值为 1024，定时时间为 1ms，并使能SysTick以及其异常请求，最后再开启SysTick定时器。

```
    jd_asm_systick_init        PROC      ;systick初始化，hal库已经初始化，这里不调用
                               EXPORT jd_asm_systick_init
                               CPSID i ;关中断
                               LDR R0, =JD_SYSTICK_CTRL ; 加载JD_SYSTICK_CTRL的地址
                               MOV R1, #0
                               STR R1, [R0] ; 先停止SysTick，以防意外产生异常请求
                               LDR R1, =0x3FF ; 让SysTick每 1024 周期计完一次。
                               STR R1, [R0,#4] ; 写入重装载的值
                               STR R1, [R0,#8] ; 往STCVR中写任意的数，以确保清除COUNTFLAG标志
```

```
MOV R1, #0x7 ; 选择FCLK作为时钟源，并使能SysTick及其异常请求
STR R1, [R0] ; 写入数值，开启定时器
CPSIE i ;开中断
BX LR
ENDP
```

值得注意的是我们使用的HAL库已经为我们初始化了SysTick，并且定时时间为 1ms，所以上述代码在我们使用HAL库时不调用，但是移植到标准库时就需要调用以上初始化代码。

我们在stm32f1xx_it.c文件找到SysTick_Handler中断处理程序。

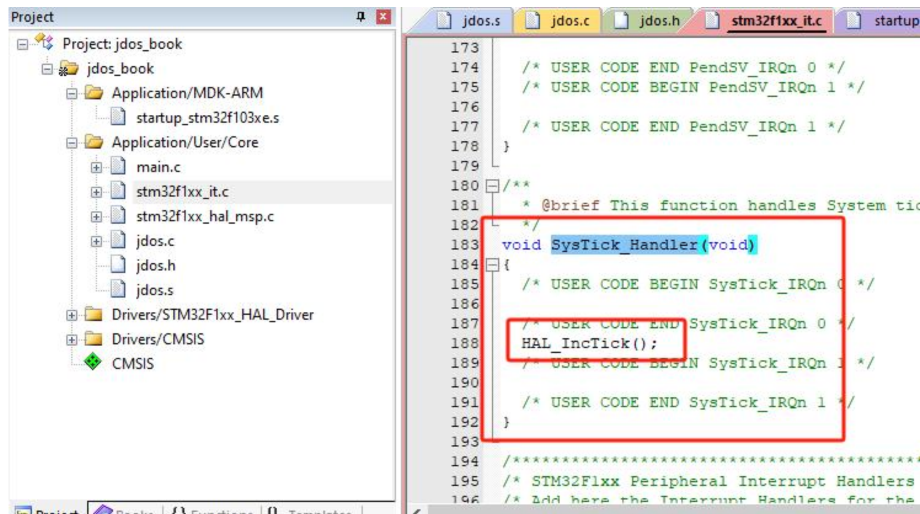

图 34 SysTick异常处理程序绑定

可以发现在SysTick_Handler中断处理程序中调用了HAL_IncTick这个函数，我们查看一下这个函数。

图 35 HAL_IncTick函数

我们发现这个函数定义成了一个weak类型，表示如果在其他地方定义了这个函数，这个地方的定义就无效，其中只执行了一行时钟累计的函数，我们将这个函数重新定义到我们jdos.c文件中，让SysTick_Handler最终调用我们jdos的处理

代码进行任务切换的工作。

```
/*hal库已自动使能systick，以下为hal库systick异常回调函数*/
void HAL_IncTick(void)
{
	uwTick += uwTickFreq;  //HAL库自带不可删除,否则hal_delay等hal库函数不可用 （1）
	jd_time++; //jd_lck++ （2）
	//扫描所有任务，将延时完成的任务更改为就绪状态，当前任务改为就绪状态，下次直接执行
	jd_task_sp->status = JD_TASK_READY;//（3）
	static struct jd_task *jd_task_temp;//（4）
	jd_task_temp = jd_task_sp->next;//（5）
	while(jd_task_sp!=jd_task_temp)//（6）
	{

	if(jd_task_temp->status==JD_TASK_DELAY&&jd_task_temp->timeout==jd_time)//
（7）
	{
		jd_task_temp->status = JD_TASK_READY;//（8）
		jd_task_temp->timeout = 0;//（9）
	}
	jd_task_temp = jd_task_temp->next;//（10）
	}
	jd_task_switch(); //jd_task_switch（11）
}
```

在上述代码（1）HAL库自带不可删除,否则hal_delay等hal库函数不可用；

（2）jdos系统时钟计数；

（3）将当前正在运行的任务更改为就绪状态，等下次调度运行；

（4）定义一个静态的临时任务控制块；

（5）将下一个任务任务控制块加载到临时控制块中；

（6）while循环，遍历整个循环链表跳出循环；

（7）查找延时任务并且延时任务的延时时间到达；

（8）将延时时间到达的任务状态更改为就绪状态；

（9）将延时时间到达的任务的延时时间进行重置；

（10）临时任务控制块切换下一个任务控制块；

（11）调用上下文切换函数。

在SysTick异常处理程序中，我们对所有任务进行遍历，将延时完成的任务的延时状态更改为就绪状态，最后我们调用上下文切换函数。

6.5 任务就绪

接下来，我们需要改变任务的状态，将默认的暂停状态更改为就绪状态，告诉系统任务已经准备就绪，等待系统调度运行。

```
/*任务就绪
*     jd_task:任务节点指针
* return: 返回JD_OK或JD_ERR
*/
int jd_task_run(struct jd_task *jd_task)
{
      if(jd_task==JD_NULL)return JD_ERR;
      jd_task->status = JD_TASK_READY; //将任务更改为就绪状态
      return JD_OK;
}
```

6.6 内核第一次运行

上面我们已经将jdos的基础搭建完成，现在我们来让内核运行起来，我们先定义一个jd_main函数来暂时做我们的空闲任务，同时在jd_main中创建 3 个测试任务。

```
//测试任务
void task1()
{
      while(1)
      {
      jd_delay(100);
      HAL_GPIO_TogglePin(GPIOC,GPIO_PIN_7);
      };
}
void task2()
{
      while(1)
      {
      jd_delay(150);
      HAL_GPIO_TogglePin(GPIOC,GPIO_PIN_7);
      };
}
void task3()
{
      while(1)
```

```
        {
        jd_delay(80);
        HAL_GPIO_TogglePin(GPIOC,GPIO_PIN_7);
        };
    }
    /*系统main,系统第一个任务，不可使用jd_task_delete删除，可添加其他任务初始化代码*/
    __weak void jd_main(void)
    {
        //printf("jd hello\r\n");
        struct jd_task *test_task1 = jd_task_create(task1,512);
        if(test_task1!=JD_NULL)jd_task_run(test_task1);
        struct jd_task *test_task2 = jd_task_create(task2,512);
        if(test_task1!=JD_NULL)jd_task_run(test_task2);
        struct jd_task *test_task3 = jd_task_create(task3,512);
        if(test_task1!=JD_NULL)jd_task_run(test_task3);
        while(1)
        {
                //jd_task_switch();
                HAL_GPIO_TogglePin(GPIOC,GPIO_PIN_7);
                // 注意此处调用延时切换任务，如果所有任务都不为就绪状态，程序将在
jd_task_switch函数中死循环，直到有就绪任务才会切换
                // 应该在此处休眠或者其他不重要的工作
                HAL_Delay(500);
        };
    }
```

完成jd_main空闲任务的创建，接下来对jdos进行初始化设置，在jdos初始化中创建一个空闲任务，将空闲任务入口设置为jd_main函数，同时初始化将全局任务控制块指针，将必要的栈指针信息传递到全局栈指针中，最后调用jd_asm_task_first_switch来直接进入jd_main函数。

```
    /*jd初始化*/
    int jd_init(void)
    {
        struct jd_task *jd_new_task = NULL;  //创建一个任务节点指针
        jd_new_task = jd_task_create(jd_main,JD_DEFAULT_STACK_SIZE);
        while(jd_new_task==NULL); //空闲任务不能创建则死循环
        jd_new_task->previous = jd_new_task; //第一个任务，指向自己
        jd_new_task->next = jd_new_task; //第一个任务，指向自己
        jd_new_task->status = JD_TASK_READY; //任务就绪
        jd_new_task->stack_sp =
(jd_new_task->stack_origin_addr+JD_DEFAULT_STACK_SIZE)&0xfffffffc; //修改指针指向栈顶
        jd_task_sp = jd_new_task; //指针移动到当前节点
        jd_task_sp_frist = jd_task_sp; //记录第一个节点
```

```
        //jd_asm_systick_init(); //启动systick,hal库已自动使能systick
        jd_asm_task_first_switch(&jd_task_sp->stack_sp,jd_main);; //进入空闲任务
        return JD_OK;
    }
```

我们将jdos初始化函数添加到main函数中while循环的前面。

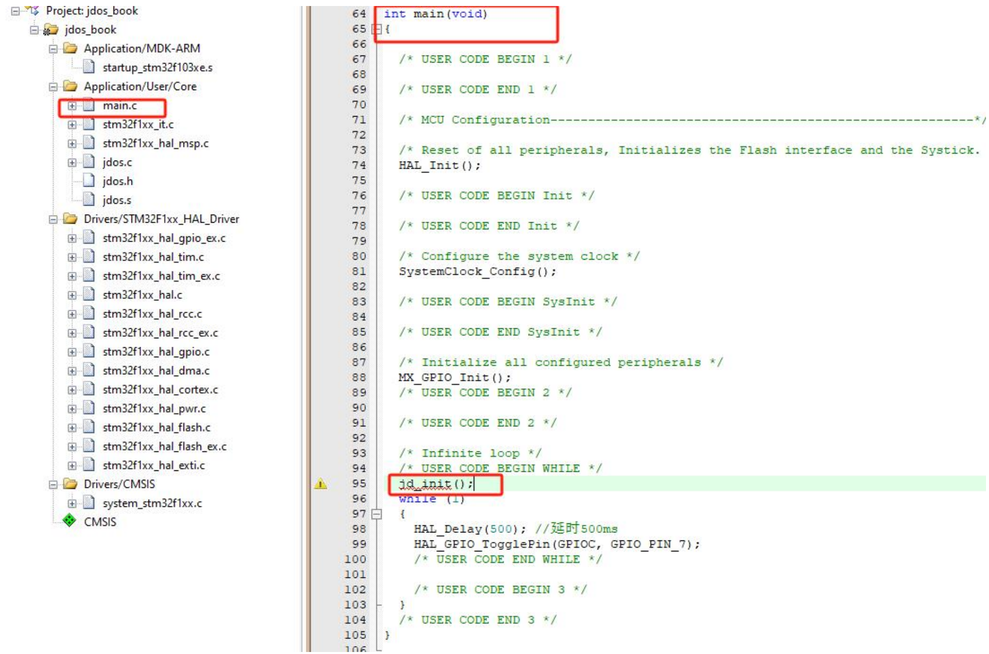

图 36 添加jd_init函数

需要调整启动文件中设定的默认堆大小，这里的堆大小指的是标准版malloc函数所能使用的堆内存的最大值，鉴于我们之前已经多次进行了空间申请，这些申请的总量已经超出了默认配置的堆大小限制，因此必须对这一参数进行修改，否则内存申请将会失败。默认堆大小也可以在STM32CubeMX中进行修改，这里我直接在启动文件中进行修改。

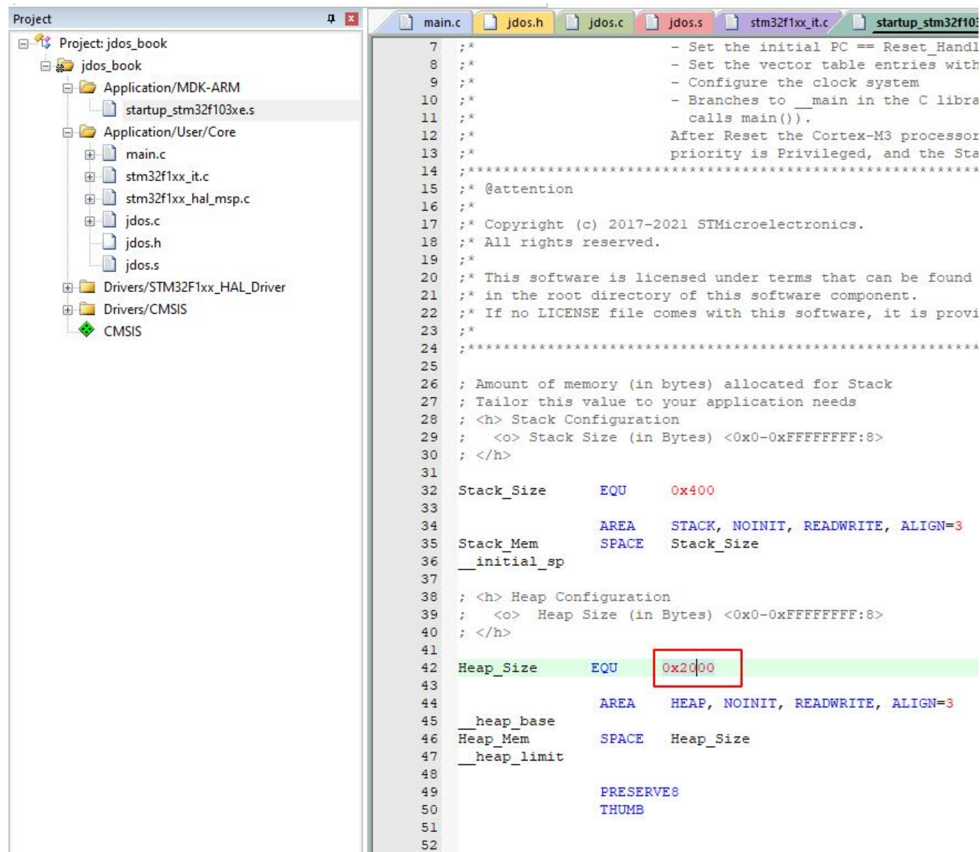

图 37 修改堆大小

在启动文件中跳转至main函数之前，我们必须关闭中断响应，由于HAL库会自动进行初始化，而这一过程可能会引发不可预测的中断响应，在jdos尚未完成初始化的情况下，响应中断可能会导致不稳定或不可预测的行为。

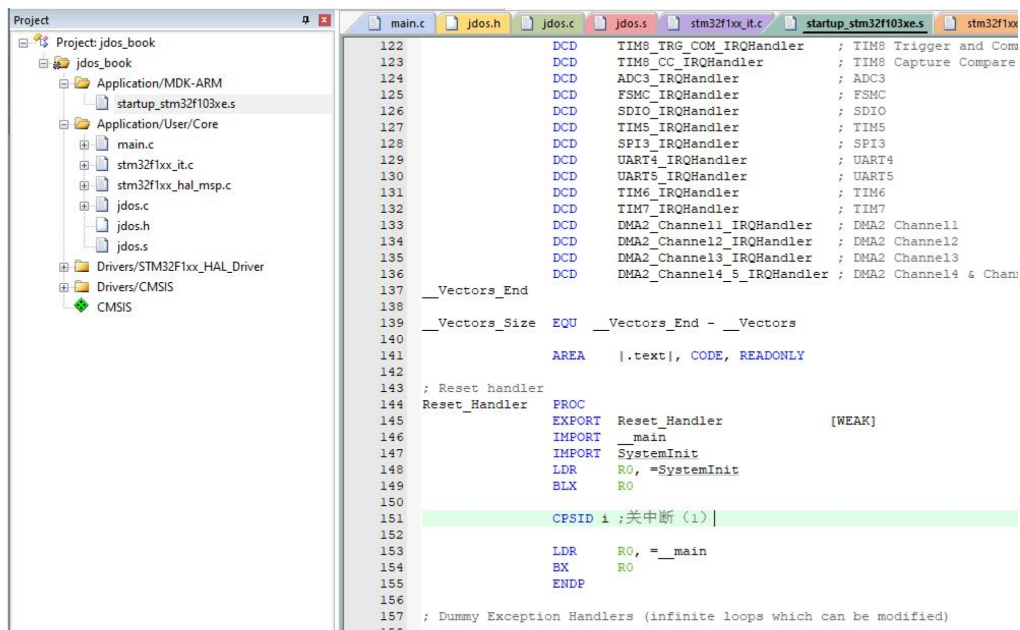

图 38 关闭中断响应

编译代码并下载至开发板后，可以观察到LED灯以快速且无规律的方式闪烁，这表明多任务切换已经成功完成。

6.7 任务延时

在上面的任务代码中，我们采用了HAL_Delay函数以实现延时效果，该函数属于HAL库的一部分，它会阻塞程序的运行。在任务执行过程中，若需要延时等待，我们期望CPU能够在此期间处理其他任务，而不是持续占用CPU资源，这样可以避免降低CPU的使用效率，所以当任务想要延时等待时，应该将当前的任务状态标记为延时状态，同时记录下延时的时间，然后让出CPU使用权切换至下一个任务，这也就是任务主动切换事件。

```
/*jdos延时，让出CPU使用权
* ms:延时时间，单位ms
*/
void jd_delay(unsigned long ms){
	if(ms==0)return;
	jd_task_sp->status = JD_TASK_DELAY;
	jd_task_sp->timeout = jd_time+ms; //将延时时间写入节点
	jd_task_switch(); //切换线程，让出CPU，等延时后调度
}
```

在上面的任务代码中，我们将所有测试任务中的HAL_Delay替换为jd_delay，然而，需要注意的是，在jd_main函数中，HAL_Delay不能被替换为jd_delay，正如前文所述，jd_main作为我们的空闲任务，必须满足jdos中至少一个任务的要求。否则，系统将无任务可执行，导致在任务切换过程中陷入死循环。

在对代码进行修改并重新编译后，将代码下载至开发板，可以观察到LED灯以快速且无规律的方式闪烁，这验证了任务延时功能的正确实现。

6.8 任务暂停与删除

在执行某些任务时，我们可能希望暂停或删除任务。为此，我们决定增加任务暂停和任务删除函数。

要暂停任务，仅需将任务状态更新为暂停即可。而删除任务，则需将其从全局任务链表中移除，并且使用free函数释放其占用的内存。

```
/*任务暂停
```

```
*     jd_task:任务节点指针
* return: 返回JD_OK或JD_ERR
*/
int jd_task_pause(struct jd_task *jd_task)
{
      if(jd_task==JD_NULL)return JD_ERR;
      jd_task->status = JD_TASK_PAUSE; //将任务更改为就绪状态
      return JD_OK;
}
/*删除任务
* jd_task:任务节点指针
* return: 返回JD_OK或JD_ERR
*/
int jd_task_delete(struct jd_task *jd_task){
      if(jd_task==JD_NULL)return JD_ERR;
      struct jd_task *jd_task_previous = jd_task->previous;
      struct jd_task *jd_task_next = jd_task->next;
      if(jd_task==jd_task_sp_frist)return JD_ERR; //判断是否为系统第一个节点，系统第
一个节点不可删除
      jd_task_previous->next = jd_task_next; //上一个节点的next指向下一个节点
      jd_task_next->previous = jd_task_previous; //下一个节点的previous指向上一个节
点
      free((unsigned long *)jd_task->stack_sp); //释放当前节点的堆栈内存
      free(jd_task); //释放当前节点内存
      return JD_OK;
}
```

6.9 外部使用

为了我们的函数能够在外部使用，我们将上面全部的函数在jdos.h中进行声明。

```
/*第一次进入任务*/
extern void jd_asm_task_first_switch(unsigned long*,void*);
/*切换任务节点，悬挂PendSV异常，PendSV中进行上下文切换*/
extern void jd_asm_pendsv_putup(void);
/*PendSV切换上下文*/
extern void jd_asm_pendsv_handler(void);
/*systick初始化*/
extern void jd_asm_systick_init(void);
/*除能 NMI 和硬 fault 之外的所有异常*/
extern void jd_asm_cps_disable(void);
/*使能中断*/
extern void jd_asm_cps_enable(void);
```

```c
/*jdos main*/
void jd_main(void);
/*jdos 系统初始化*/
int jd_init(void);
/*jdos延时，让出CPU使用权*/
void jd_delay(unsigned long ms);
/*创建任务*/
struct jd_task *jd_task_create(void (*task_entry)(),unsigned int stack_size);
/*更改为就绪状态，等待调度*/
int jd_task_run(struct jd_task *jd_task);
/*删除任务，释放内存*/
int jd_task_delete(struct jd_task *jd_task);
/*暂停任务*/
int jd_task_pause(struct jd_task *jd_task);
/*手动进行任务调度*/
void jd_task_switch(void);
```

小结

本章详细阐述了jdos内核的编程实现，涵盖了任务的创建、状态转换、堆栈管理以及时间片轮转等核心机制。通过设计任务控制块和任务链表，我们实现了在jdos系统中如何维护任务信息与状态，并且通过利用时间片轮转调度来实现多任务的并发执行。

第七章 任务抢占式

任务抢占式是一种提高系统实时性能的重要机制，它允许高优先级的任务在低优先级任务正在执行时抢占CPU资源，从而确保关键任务能够及时响应。目前在jdos系统中，我们通过循环的方式来执行任务，现在我们通过设置任务优先级来实现任务的抢占式调度，当一个高优先级任务准备就绪时，系统会自动保存当前任务的状态，并切换到高优先级任务执行，完成高优先级任务后，系统再恢复之前任务的状态，继续执行，这样的机制大大增强了系统的响应速度和任务执行的灵活性。

7.1 旧任务管理流程分析

在设计新的任务管理流程之前，我们需对现有的任务管理流程进行分析。根据现行的代码逻辑，所有任务均被存储在一个链表中，这种做法存在一个明显的缺陷：当任务中处于延时状态的数量增多时，内核不得不频繁地搜索就绪状态的任务。此外，每当系统时钟事件触发时，内核必须遍历整个链表以确定每个任务的延时是否已经结束。这两种情况共同作用，大大导致了系统效率的下降。

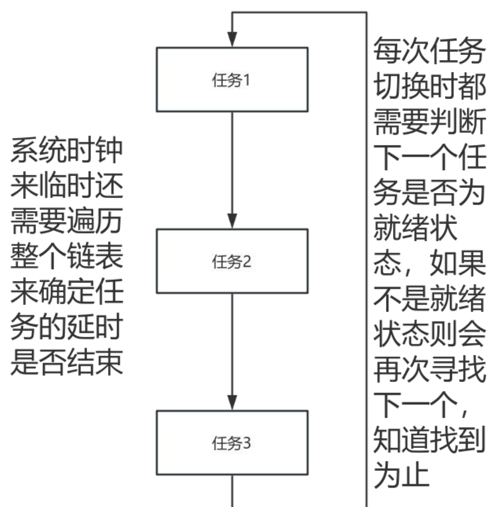

图 39 旧任务管理流程示意图

为了满足任务的优先级需求，我们必须对任务进行排序，确保高优先级的任务总是位于队列的最前端，以符合抢占式调度的要求。同时，我们还需要根据任

务的延时时间进行排序，这样一来，在系统时钟事件触发时，我们仅需检查第一个任务的延时是否已经到达，从而避免每次都需要遍历整个链表的低效操作。

然而，旧的循环链表设计，将无法同时满足优先级排序和延时排序的需求，强行在此基础上进行设计会引起逻辑混乱，使得代码难以维护。因此，我们必须将这两种排序机制分开处理。

7.2 新任务管理流程设计

我们定义两个双向链表：一个用于表示任务就绪链表，另一个表示任务延时链表，将就绪任务与延时任务分离，就绪链表顶端的任务始终处于运行状态，所有处于其他状态的任务，我们统称为任务池中的任务。

关于任务切换的条件，我们之前已经实现了两种：系统时钟事件和任务主动切换事件，这些事件分别通过SysTick和jd_delay来实现。然而，任务结束事件尚未实现，我们将其纳入重新设计的任务管理流程中。

任务池中的任务能够被配置为定时任务，这些定时任务包括循环定时任务和一次性定时任务。它们随后被添加至延时链表，或者直接转换为就绪状态并加入到就绪链表中。在就绪链表中，任务根据其优先级进行排序，优先级最高的任务将获得CPU的使用权，一旦任务执行完毕，产生任务结束事件，它将被重新加入到任务池中。如果任务执行出现延时，或者被设置为循环定时任务，它将被加入到延时链表中。在延时链表中，任务根据它们的延时时间长短进行排序，每当系统时钟事件触发时，系统会比较延时链表顶端的任务，一旦该任务的延时完成，它将被转移到就绪链表中。当然，延时任务也可以选择暂停执行，所有任务除系统空闲任务外都可以进行销毁，由系统进行资源回收。

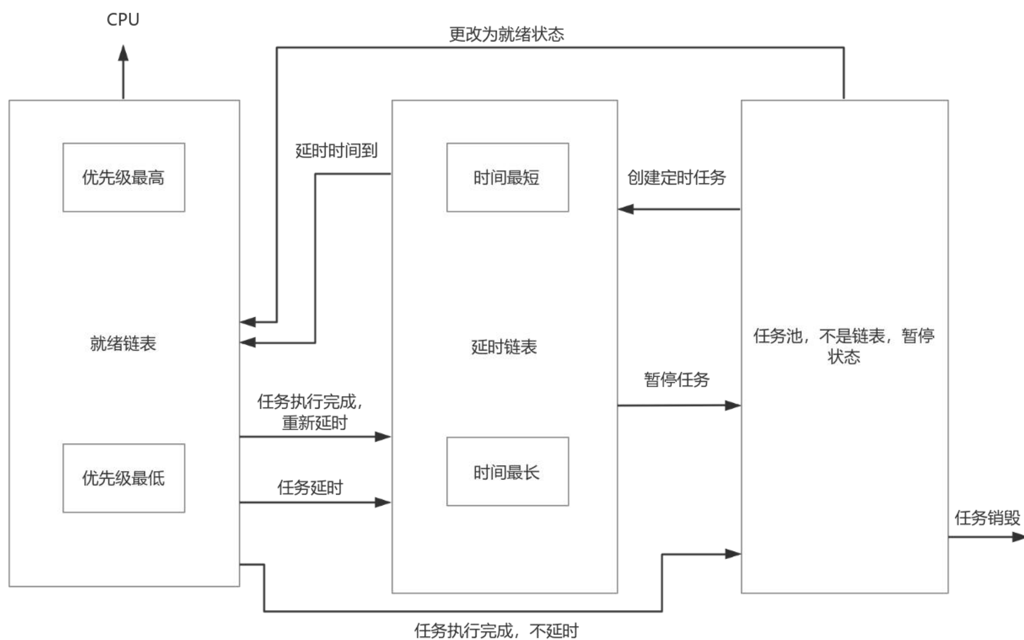

图 40 新任务管理流程

7.3 头文件代码修改

在重新设计任务管理流程之前，我们先在jdos.h中优化一下关键字名称，方便后续的代码维护。

```
// jdos变量重新定义
typedef unsigned char jd_uint8_t;
typedef char jd_int8_t;
typedef unsigned short jd_uint16_t;
typedef signed short jd_int16_t;
typedef unsigned int jd_uint32_t;
typedef signed int jd_int32_t;
typedef unsigned long jd_uint64_t;
typedef signed long jd_int64_t;
typedef jd_uint32_t jd_time_t;
```

同时将所有的结构体的名字都重新定义，简化代码。

```
/******************结构体定义************************/
/*枚举任务状态*/
typedef enum jd_task_status
{
	JD_TASK_READY = 0, // 任务就绪状态
	JD_TASK_RUNNING,   // 任务运行状态
```

```
        JD_TASK_DELAY,     // 任务延时状态
        JD_TASK_PAUSE,     // 任务暂停状态
} jd_task_status_t;
/*定义所有寄存器，根据入栈规则有先后顺序*/
typedef struct all_register
{
        // 手动入栈
        jd_uint32_t r4;
        jd_uint32_t r5;
        jd_uint32_t r6;
        jd_uint32_t r7;
        jd_uint32_t r8;
        jd_uint32_t r9;
        jd_uint32_t r10;
        jd_uint32_t r11;
        // 自动入栈
        jd_uint32_t r0;
        jd_uint32_t r1;
        jd_uint32_t r2;
        jd_uint32_t r3;
        jd_uint32_t r12;
        jd_uint32_t lr;
        jd_uint32_t pc;
        jd_uint32_t xpsr;
} all_register_t;
```

根据上面的新任务管理流程，我们对定时任务的状态和任务自动销毁状态进行枚举。

```
/*定时任务使用状态*/
typedef enum jd_timer_status
{
        JD_TIMER_NOTIMER = 0, // 不是定时器任务
        JD_TIMER_NOLOOP,      // 是定时任务，但不是循环定时任务
        JD_TIMER_LOOP,        // 是循环定时任务
} jd_timer_status_t;
/*任务自动销毁状态*/
typedef enum jd_task_auto_delate
{
        JD_TASK_NODELATE = 0, // 不销毁任务，不回收内存，下次可不用从新create
        JD_TASK_DELATE,       // 系统销毁任务，回收内存
} jd_task_auto_delate_t;
```

鉴于我们需要维护多个链表，因此必须对链表操作进行抽象化处理，以实现统一管理。这样一来，无论何时何地需要链表控制，只需在相应的结构体中嵌入

链表结构体即可。

这里需要特别说明的是：在C语言中，所有的语句都与地址相关，以jd_task结构体为例，它的地址实际上就是其内部第一个元素的地址。基于这一特性，我们将链表结构体设置为jd_task的第一个内部元素，当处理任务链表时，我们只需将任务控制块的地址转换为链表结构体的地址即可。

我们在任务控制块中添加上新的链表结构、任务的优先级、任务的定时使用状态、任务的循环定时时间以及任务自动销毁状态。

```
// 定义链表节点
typedef struct jd_node_list
{
      struct jd_node_list *previous; // 上一个节点
      struct jd_node_list *next;     // 下一个节点
} jd_node_list_t;
/*定义任务控制块*/
typedef struct jd_task
{
      jd_node_list_t node;              // 链表节点
      void (*entry)();                  // 指向任务入口函数
      jd_task_status_t status;          // 当前任务状态
      jd_uint32_t stack_size;           // 堆栈大小
      jd_uint32_t stack_sp;             // 堆栈指针
      jd_uint32_t stack_origin_addr;    // 堆栈起始地址
      jd_uint32_t timeout;              // 延时溢出时间, 单位ms, 为0 则没有延时
      jd_int8_t priority;               // 优先级-128 - 127,越低优先级越高,一般从0
开始用
      jd_timer_status_t timer_loop;     // 是否为定时任务,如果是定时任务是否为循环模
式
      jd_uint32_t timer_loop_timeout;   // 任务处于循环状态, 定时时间
      jd_task_auto_delate_t auto_delate; // 任务执行完成后是否需要系统销毁任务
} jd_task_t;
```

7.4 抽象链表操作

我们将链表的所有操作进行抽象化处理。链表操作主要包括插入节点和删除节点。下面展示的是插入节点的示意图，图中实线箭头表示原始链表的连接方式，节点 1 与节点 3 原本是相连的。现在，我们在节点 1 和节点 3 之间插入了一个新的节点 2，虚线箭头展示了插入节点 2 后的新的连接方式，即节点 1 连接到节点 2，节点 2 再连接到节点 3。

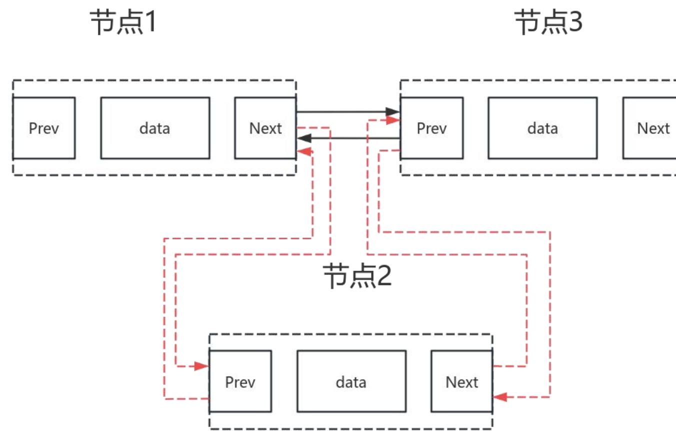

图 41 插入节点示意图

代码实现如下，其中包含了对无效节点的检查，以防止空指针异常导致系统崩溃，同时针对表头和表尾的位置采用了不同的连接策略。

```
/**
* @description: 新节点插入链表中
* @param {jd_node_list_t} *node_previous 想要插入的链表节点处的上一个节点
* @param {jd_node_list_t} *node 想要插入的节点,为JD_NULL表示连接前后两个节点
* @param {jd_node_list_t} *node_next 想要插入的链表节点处的下一个节点
* @return {*}
*/
jd_int32_t jd_node_insert(jd_node_list_t *node_previous, jd_node_list_t *node,
jd_node_list_t *node_next)
{
// 传入节点无效，无法插入
if (node_previous == JD_NULL && node_next == JD_NULL)
{
return JD_ERR;
}
// 连接前后两个节点
else if (node == JD_NULL)
{
// 前一个节点为空
if (node_previous == JD_NULL)
{
node_next->previous = JD_NULL;
}
// 后一个节点为空
else if (node_next == JD_NULL)
{
node_previous->next = JD_NULL;
}
```

```
// 连接两个节点
else
{
node_next->previous = node_next;
node_previous->next = node_next;
}
}
// 当前插入节点为表头
else if (node_previous == JD_NULL)
{
node->next = node_next; // 新节点指向下一个节点
node->previous = JD_NULL;         // 新节点指向上一个JD_NULL
node_next->previous = node; // 下一个节点指向新节点
}
// 当前插入节点为表尾
else if (node_next == JD_NULL)
{
node_previous->next = node;                // 上一个节点指向新节点
node->previous = node_previous; // 新节点指向上一个节点
node->next = JD_NULL;    // 新节点指向JD_NULL
}
// 当前插入节点为表中
else
{
node->next = node_next;          // 新节点指向下一个节点
node->previous = node_previous; // 新节点指向上一个节点
node_previous->next = node;                // 上一个节点指向新节点
node_next->previous = node;                // 下一个节点指向新节点
}
return JD_OK;
}
```

移除节点意味着将特定节点从链表中彻底删除，通过将该节点的前一个节点与后一个节点直接相连来实现，以下代码中我们同样需要对表头和表尾进行判断，已防止破坏原本的链表。

```
/**
* @description: 删除节点
* @param {jd_node_list_t} *list 需要进行删除的链表
* @param {jd_node_list_t} *node 需要删除的节点
* @return {*}
*/
jd_node_list_t *jd_node_delete(jd_node_list_t *list, jd_node_list_t *node)
{
if (list == JD_NULL || node == JD_NULL)
```

```
return JD_NULL;
// 判断节点是否在表头
if (node == list)
{
// 移动表头
list = list->next;
// 如果移动后表头不为空
if (list != JD_NULL)
list->previous = JD_NULL;
}
// 判断是否在表尾
else if (node->next == JD_NULL)
{
// 删除最后一个节点
node->previous->next = JD_NULL;
}
// 在表中
else
{
jd_node_insert(node->previous, JD_NULL, node->next);
}
// 清空节点信息
node->previous = JD_NULL;
node->next = JD_NULL;
return list;
}
```

我们已将链表操作简化为两个核心函数，在新的任务管理系统中，设计两个任务链表：任务就绪链表和任务延时链表。这两个链表由统一的管理系统进行管理，以下是任务优先级和任务延时时间比较的代码实现。此外，我们需要重新定义全局任务变量，全局就绪任务链表、全局延时任务链表、全局任务运行时指针、全局任务堆栈指针、全局下一个任务的堆栈指针、全局系统空闲任务指针、以及全局任务的入口和退出入口。

```
jd_node_list_t *jd_task_list_readying = NULL; // 创建就绪任务链
jd_node_list_t *jd_task_list_delaying = NULL; // 创建延时任务链表
jd_task_t *jd_task_runing = NULL;// 创建当前任务指针
void *jd_task_stack_sp = NULL; // 创建当前任务堆栈指针的地址
void *jd_task_next_stack_sp = NULL;// 创建下一个任务堆栈指针的地址
jd_task_t *jd_task_frist = NULL;// 创建一个系统空闲任务
jd_uint32_t jd_task_entry;        // 任务入口
jd_uint32_t jd_task_exit_entry; // 任务exit入口
/**
```

```
 * @description: 比较函数，用于jd_node_in_rd中使用
 * @param {jd_task_t} *task1 用于比较的任务1
 * @param {jd_task_t} *task2 用于比较的任务2
 * @return {*}
 */
jd_int64_t compare_priority(jd_task_t *task1, jd_task_t *task2){
	return task1->priority - task2->priority;
}
jd_int64_t compare_timeout(jd_task_t *task1, jd_task_t *task2){
	return task1->timeout - task2->timeout;
}
```

我们编写节点插入代码，将任务节点插入到相应的链表中，具体取决于链表的类型。

```
/**
 * @description: 将节点插入就绪或者延时链表
 * @param {jd_node_list_t} *list 要插入的链表
 * @param {jd_node_list_t} *node 要插入的节点
 * @return {*}
 */
jd_node_list_t *jd_node_in_rd(jd_node_list_t *list, jd_node_list_t *node){
	jd_task_t *jd_task_temp, *jd_task_in_temp;
	jd_node_list_t *node_temp;
// 链表没有任务
	if (list == JD_NULL)
	{
		list = node;
		list->next = JD_NULL;
		list->previous = JD_NULL;
	} //（1）
	// 链表中有任务
	else
	{
		// 比较函数选择
		jd_int64_t (*compare)(jd_task_t *task1, jd_task_t *task2);//（2）
		if (list == jd_task_list_readying)
		{
		compare = compare_priority;
		}//（3）
		else
		{
			compare = compare_timeout;
		}//（4）
		// 临时节点，用于遍历链表
```

```
                node_temp = list; //（5）
                // 插入节点的任务数据
                jd_task_in_temp = (jd_task_t *)node; //（6）
                // 遍历链表
                while (1)
                {
                        jd_task_temp = (jd_task_t *)node_temp; // 获取任务数据（7）
                        // 如果数字越小，优先级越高，或者延时时间越短
                        if (compare(jd_task_in_temp, jd_task_temp) <= 0)//（8）
                        {
                        // 判断为表头
                                if (node_temp->previous == JD_NULL)
                                {
                                        list = node; // 切换表头
                                        jd_node_insert(JD_NULL, list,
node_temp);
                                }//（9）
                                else
                                {
                                        jd_node_insert(node_temp->previous,
node, node_temp);
                                }//（10）
                                break;//（11）
                        }
                        // 判断表尾
                        if (node_temp->next == JD_NULL)
                        {
                                // 将任务插入到表尾
                                jd_node_insert(node_temp, node, JD_NULL);
                                break;
                        }//（12）
                        // 临时节点切换为下一个节点
                        node_temp = node_temp->next;//（13）
                }
        }
        return list;//（14）
    }
```

在上述代码中（1）判断传入的链表是否为空，如果为空，则将传入的节点作为链表的第一个节点；

（2）定义一个名为compare的函数指针，用于选择是优先级比较还是延时比较函数；

（3）如果传入的链表是就绪链表，则选择优先级比较函数；

（4）如果传入的是延时链表，则选择延时比较函数；

（5）创建一个临时节点；

（6）将传入的节点转换成任务控制块；

（7）将传入的链表中的节点转换成任务控制块；

（8）将传入的节点与链表中的节点进行优先级比较或延时比较，若传入节点的优先级更高或其延时较短，则执行链表插入操作；

（9）判断当前进行比较的链表节点是否为表头，是表头则执行表头插入逻辑；

（10）不为表头，则执行正常插入节点插入逻辑；

（11）已经插入节点，所以跳出while循环；

（12）如果链表节点已经到达表尾，则将节点插入到表尾，同时跳出while循环；

（13）切换到链表当前节点的下一个节点；

（14）将插入完成的链表进行返回。

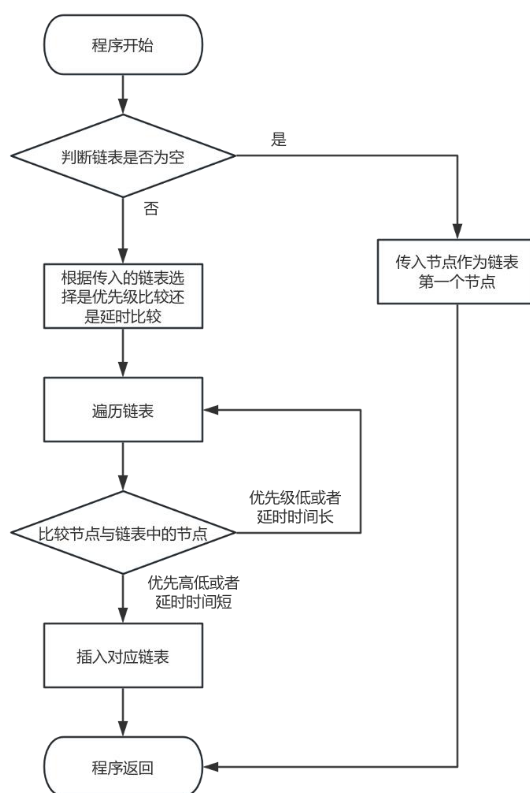

图 42 就绪与延时链表插入示意图

7.5 任务结束事件与主动切换事件

在旧的任务切换机制中，我们通过触发PendSV异常来主动切换任务事件。然而，依赖PendSV进行上下文切换存在一些问题，由于PendSV异常不会立马引发中断响应，这可能导致任务上下文切换不够精确，在大多数情况下，任务执行期间主动要求上下文切换都需要精确的执行，以确保系统的稳定性。因此，在这种情况下，我们应当采用SVCall异常来实现更精确的切换。

通过SVCall来触发任务结束事件与主动切换事件，SVCall在执行过程中会自动携带一个特定的异常号，这个异常号仅与SVCall的使用相关，与前面描述的异常号并无关联，我们可以在任务中通过不同的SVCall异常号来选择SVCall不同的中断处理程序。

打开jdos.s，编写任务结束事件的中断响应代码，在任务结束事件中我们需要对任务的一些数据进行重置，以确保下次任务重新运行时可以正确启动。

```
jd_asm_task_exit_switch PROC     ;任务结束运行（没有while），切换下一个任务
                        EXPORT  jd_asm_task_exit_switch
                        CPSID i ;关中断
               ; 此处应该硬性改变堆栈中LR与PC的，相当于回复第一次执行的程序入口数据
                        LDR R1,=jd_task_entry ;此次程序的入口地址
                        LDR R1,[R1] ;（1）
                        LDR R2,=jd_task_exit_entry ;此次程序的exit的入口地址
                        LDR R2,[R2] ;（2）
                        LDR R0,=jd_task_stack_sp ;此次任务的栈指针
                        LDR R0,[R0]
                        LDR R0,[R0] ;(3)
                        STR R1,[R0,#56] ;(4)
                        STR R2,[R0,#52] ;(5)
                        ;取下一个任务的堆栈指针,恢复现场
                        LDR R1,=jd_task_next_stack_sp
                        LDR R1,[R1]
                        LDR R0,[R1] ;(6)
                        LDMFD R0!,{R4-R11} ;(7)
                        CPSIE i ;开中断
                        BX LR ;(8)
                        ENDP
```

上述代码（1）加载当前任务的程序入口地址到R1；

（2）加载当前任务退出函数入口地址到R2；

（3）加载当前任务的栈指针到R0；

（4）将当前任务的程序入口地址加载到栈指针+56 字节后的 4 字节空间，我们打开jdos.h文件查看所有寄存器的定义，可以发现恰好加载到所有寄存器中的PC程序计数器中，下次重新启动任务时，直接出栈将任务的程序入口地址装载到PC程序计数器中，完成任务启动；

（5）将当前任务的退出函数入口地址加载到栈指针+52 字节后的 4 字节空间，恰好是LR链接寄存器中，任务结束后通过LR链接寄存器中的地址返回到任务结束处理函数中，此时，可以在此函数中触发任务结束事件。

（6）取下一个任务的栈指针；

（7）手动出栈寄存器R4-R11；

（8）切换下一个任务。

任务结束事件的中断处理程序已经完成，接下来实现SVCall异常处理的程序，在SVCall异常中我们需要通过判断SVCall的异常号来确定任务结束事件和任务出动切换事件。

```
jd_asm_svc_handler      PROC     ;SVC处理
                        EXPORT  jd_asm_svc_handler
                        TST LR, #0x4 ; 测试EXC_RETURN的比特2 （1）
                        ITE EQ ; 如果为0
                        MRSEQ R0, MSP ;
                        MRSNE R0, PSP ; （2）
                        ;  获取返回地址 (原理是与发生异常时硬件压栈的顺序相关)
       ;  这里获得返回地址的原因是为了定位产生异常前执行的最后一条指令，也就是SVC指令
                        LDR R1, [R0, #24] ;（3）
                        ; 获取SVC指令的低 8 位，也就是系统调用号，返回地址的上一条就是
SVC指令，获取的是机器码
                        LDRB R1, [R1, #-2] ;（4）
                        ;svc_0 服务，任务自动切换下一个任务
                        CMP R1, #0
                        BEQ svc_handler_0 ;（5）
                        ;svc_1 服务，任务结束
                        CMP R1, #1
                        BEQ svc_handler_1 ;（6）
                        BX LR ;（7）
```

在上述代码中（1）测试寄存器LR的第 2 位（从 0 开始计数）。#0x4 是立即数 4，表示测试LR寄存器的第 2 位（因为 0x4 的二进制表示是 100）。如果LR的第 2 位是 1，则状态寄存器中的Z标志（零标志）为 1，否则设置为 0，LR的第 2 位用于区分MSP和PSP，具体参阅《DDI0403E_e_armv7m_arm》；

（2）若LR（链接寄存器）的第二位为 0，则将MSP（主堆栈指针）的栈指针值加载至R0 寄存器；若该位为 1，则加载PSP（进程堆栈指针）的栈指针值至R0 寄存器。这一操作是为了在触发SVCall异常时能够区分是使用MSP还是PSP。在处理SVCall异常时，需要知道确切的异常号，鉴于Cortex-M3 处理器具备这两个堆栈指针，因此在程序中必须明确区分MSP和PSP，以保证能够正确获取到SVCall异常号；

（3）加载栈指针+24 字节后面的 4 字节空间的数据到R0，我们打开jdos.h，查看全部寄存器的定义，找到自动入栈的定义，发现恰好加载的LR链接寄存器的数据，这里面现在保存了任务产生异常时的下一条指令的地址也就是返回地址，所以这条指令是将返回地址加载到R0 中，这里获得返回地址的原因是为了定位产生异常前执行的最后一条指令，也就是SVC指令；

（4）将R0 中的地址-2 字节后的地址中的数据加载到R1 中，也就是将产生异常前执行的最后一条指令的低 8 位加载R1 中，也就是机器码，这里为什么这样做呢？原因是SVC指令编译的机器码的低 8 位其实就是SVCall的异常号，关于SVC指令编译成机器码的详情请参阅《DDI0403E_e_armv7m_arm》，这里相当于拿到了任务执行SVC指令时所携带的异常号；

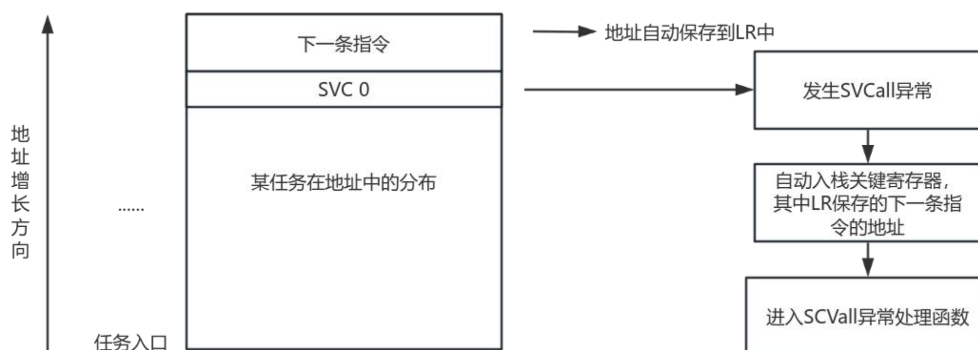

图 43 SVCall异常发生与内存示意图

（5）R1 中的值与 0 比较，相等则跳转到svc_handler_0 程序；

（6）R1 中的值与 1 比较，相等则跳转到svc_handler_1 程序；

（7）程序返回。

注意：需要将新定义的jd_asm_svc_handler函数与SVCall异常处理函数绑定，在stm32f1xx.it.c中的SVC_Handler函数中添加jd_asm_svc_handler()代码即可。

在这里我们定义svc_handler_0 是任务主动切换处理程序，svc_handler_1 为任务结束处理程序，我们将svc_handler_0 主动切换处理程序与PendSV处理程序绑定，主动切换与正常的PendSV切换是一致的，只不过这里我们不编写重复的代码，借用PendSV的处理程序。svc_handler_1 任务结束处理程序与之前编写的任务结束事件的中断响应代码绑定。

```
svc_handler_0
        B jd_asm_pendsv_handler ;正常切换
        BX LR
        ENDP
svc_handler_1
        B jd_asm_task_exit_switch
        BX LR
        ENDP
```

我们完成了SVCall异常处理程序与任务结束和任务主动切换事件的绑定，在任务中调用以下代码可以正确触发SVC异常。

```
jd_asm_svc_task_switch  PROC      ;SVC call
                        EXPORT  jd_asm_svc_task_switch
                        CPSIE i ;开中断
                        SVC 0
                        BX LR
                        ENDP
jd_asm_svc_task_exit    PROC      ;SVC call
                        EXPORT  jd_asm_svc_task_exit
                        CPSIE i ;开中断
                        SVC 1
                        BX LR
                        ENDP
```

7.6 任务结束处理

尽管我们能够通过在任务执行过程中手动调用SVC指令来引发SVCall异常，从而实现精确的上下文切换，但这种方法并不优雅，我们期望在任务运行结束后，系统能够自动介入处理，并且能够支持更多功能。

例如，是否应当调用钩子函数（目前我选择不实现它，因为它对于满足我们当前的需求并非必要），以及如何区分任务是主动结束还是被动结束，或者为任务的下一次运行做准备，又或者处理系统任务销毁等功能。简而言之，我们需要一个专门的任务结束处理程序。

```
/**
 * @description:任务退出函数,用户任务处理完后自动处理,系统自动调用
 * @return {*}
 */
void jd_task_exit()
{
        jd_task_t *jd_task = jd_task_runing;

        jd_task_entry = (jd_uint32_t)jd_task->entry;  // 传递程序入口值
        jd_task_exit_entry = (jd_uint32_t)jd_task_exit; // 传递退出时程序销毁入口

        if (jd_task->auto_delate == JD_TASK_NODELATE)
        {
                // 暂停任务
                jd_task_pause(jd_task);
        }
        else
        {
                // 删除任务
                jd_task_delete(jd_task);
        }

        jd_task->stack_sp = (jd_uint32_t)((jd_task->stack_origin_addr) +
jd_task->stack_size - sizeof(struct all_register))&0xfffffffc; // 腾出寄存器的空间
        all_register_t *stack_register = (struct all_register *)jd_task->stack_sp;

        // 设置必要数据
        stack_register->lr = (jd_uint32_t)jd_task_exit;
        stack_register->pc = (jd_uint32_t)jd_task->entry;
        stack_register->xpsr = 0x01000000L;

        jd_task->status = JD_TASK_DELAY;

        if (jd_task->timer_loop == JD_TIMER_LOOP)
                // 将节点加入延时链表
                jd_task_list_delaying = jd_node_in_rd(jd_task_list_delaying,
&jd_task_runing->node);

        jd_task_stack_sp = &jd_task->stack_sp;
        // 获取数据域
        jd_task = (jd_task_t *)jd_task_list_readying; // 获取任务数据
        // 任务暂停或延时状态，或者当前任务优先级低，当前任务放弃CPU使用权
        jd_task->status = JD_TASK_RUNNING;
// 即将运行的任务改为正在运行状态
```

```
        jd_task_runing = jd_task;
// 更改当前为运行的任务
        jd_task_next_stack_sp = &jd_task_runing->stack_sp; // 更新下一个任务全局栈指
针变量

        // 这里不是悬挂PendSV异常，所以直接跳转会出发异常，寄存器数据不会自动出栈，应该使
用SVC呼叫异常

        jd_asm_svc_task_exit();
    }
```

在上述代码中（1）定义一个任务块控制指针指向当前运行的任务；

（2）将当前任务的入口值传递给全局程序入口值；

（3）将全局任务结束入口函数更改为任务结束处理程序jd_task_exit；

（4）更改当前任务状态为暂停状态；

（5）初始化任务栈指针；

（6）将栈指针转换成所有寄存器指针；

（7）设置栈中LR的值为任务结束处理程序jd_task_exit；

（8）设置栈中PC的值为任务入口函数的地址；

（9）初始化状态寄存器；

（10）如果为定时循环状态则将任务状态更改为延时状态；

（11）将节点加入到延时链表中；

（12）将当前任务的栈指针传递给全局任务栈指针；

（13）如果是自动删除任务则删除当前任务

（14）获取就绪任务链表中优先级最高的任务控制块；

（15）将任务更改为正在运行状态；

（16）将全局正在运行任务控制块指向任务控制块；

（17）将全局下一个任务全局栈指针指向正在运行任务的栈指针；

（18）切换下一个任务。

7.7 修改任务创建函数

我们对jd_task_create这一任务创建函数进行修改，以确保之前开发的功能能够正常运作。同时，对任务栈的内存占用进行精细化管理，目的是为了提升内存

的使用效率并进一步优化系统整体的性能。

我们移除jd_request_space内存分配函数，在jd_request_space的设计初期，为了简化操作，我们分别进行了两次内存申请，使得任务控制块的内存空间与任务栈空间不是连续的，这将导致内存碎片化严重的问题，同时也不利于任务空间的管理。现在，我们将内存申请次数减少至一次，并将原本分散的任务控制块空间与栈空间合并，以优化内存使用。

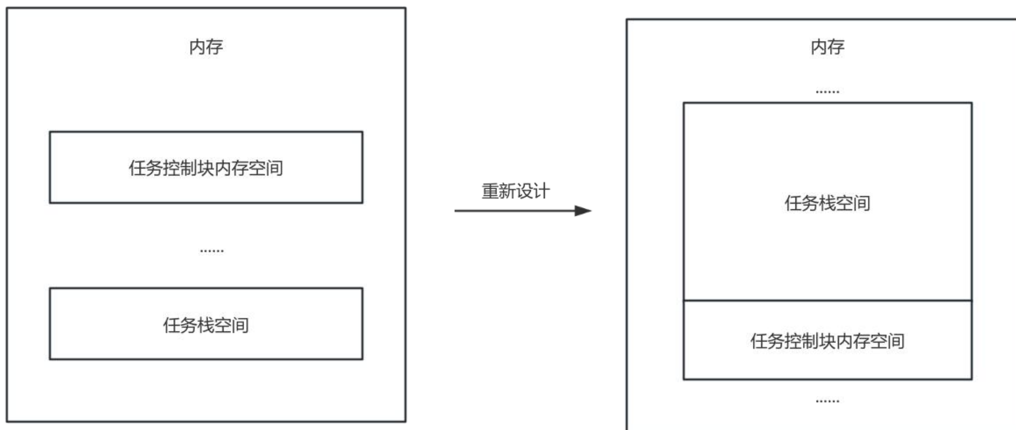

图 44 重新设计任务空间分布图

```
/**
 * @description: 创建任务
 * @param {jd_uint32_t} stack_size 任务栈大小
 * @param {jd_int8_t} priority 任务优先级-128-127，数字越小，优先级越高
 * @return {*}
 */
jd_task_t *jd_task_create(void (*task_entry)(), jd_uint32_t stack_size, jd_int8_t
priority){
    jd_task_t *jd_new_task = NULL; // 创建一个任务节点指针
    jd_new_task = (jd_task_t *)malloc(stack_size); // 分配空间
    if (jd_new_task == NULL)
        return JD_NULL; // 判断分配空间是否成功

    jd_new_task->node.next = JD_NULL;     // 初始化节点指针
    jd_new_task->node.previous = JD_NULL;          // 初始化节点指针
    jd_new_task->stack_origin_addr = (jd_uint32_t)jd_new_task; // 记录栈底指针

    jd_new_task->timeout = 0;                            // 没有延时时间
    jd_new_task->entry = task_entry;        // 任务入口
    jd_new_task->status = JD_TASK_PAUSE;               // 创建任务，状态为暂停状态，
等待启动
    jd_new_task->stack_size = stack_size; // 记录当前任务堆栈大小
```

```c
        jd_new_task->stack_sp = (jd_uint32_t)((jd_new_task->stack_origin_addr) +
jd_new_task->stack_size - sizeof(struct all_register))&0xfffffffc; // 腾出寄存器的空间
        all_register_t *stack_register = (struct all_register *)jd_new_task->stack_sp;


        // 将任务运行数据搬移到内存中
        stack_register->lr = (jd_uint32_t)jd_task_exit;
        stack_register->pc = (jd_uint32_t)jd_new_task->entry;
        stack_register->xpsr = 0x01000000L; // 由于Armv7-M只支持执行Thumb指令，因此必
须始终将其值保持为1，否则任务切换会异常

        jd_new_task->priority = priority; // 设置优先级

        jd_new_task->timer_loop = JD_TIMER_NOTIMER;     // 不是定时任务
        jd_new_task->auto_delate = JD_TASK_NODELATE; //         任务执行完成后不自动
回收内存，任务不删除，下次可直接运行

        return jd_new_task; // 返回当前任务节点
    }
```

7.8 修改任务暂停和删除函数

我们对任务暂停和任务删除这两个功能进行重新设计和实现，在旧的设计中，暂停任务的操作相对简单，仅仅是将任务的状态标记为暂停状态，而在新的设计中，我们通过就绪和延时链表对任务进行管理，所以还需要增加将任务从就绪队列或延时队列中移除的步骤，确保任务处于暂停状态。

删除任务的操作首先调用暂停任务的逻辑，确保任务处于暂停状态，接下来，我们释放与该任务相关的所有内存资源，确保系统资源得到及时回收，避免内存泄漏等问题。

同时，我们还需设计一个函数将任务自动删除的功能暴露给用户，以便实现任务结束后自动回收。

```c
    /**
     * @description: 任务暂停
     * @param {jd_task_t} *jd_task 任务节点指针
     * @return {*}
     */
    jd_int32_t jd_task_pause(jd_task_t *jd_task)
    {
```

```
        if (jd_task == JD_NULL)
                return JD_ERR;

        // 不能更改系统空闲任务状态，始终在就绪链表
        if (jd_task == jd_task_frist)
                return JD_ERR;

        // 本来就为暂停状态
        if (jd_task->status == JD_TASK_PAUSE)
                return JD_OK;

        // 在就绪链表表头
        if (jd_task_list_readying == &jd_task->node)
        {
                // 移动表头，同时将表头中指向的上一个节点信息删除
                jd_task_list_readying = jd_task->node.next;
                jd_task_list_readying->previous = JD_NULL;
        }
        // 在延时链表表头
        else if (jd_task_list_delaying == &jd_task->node)
        {
                // 移动表头，同时将表头中指向的上一个节点信息删除
                jd_task_list_delaying = jd_task->node.next;
                jd_task_list_delaying->previous = JD_NULL;
        }
        else
        {
                // 直接删除节点
                jd_node_insert(jd_task->node.previous, JD_NULL,
jd_task->node.next);
        }
        jd_task->status = JD_TASK_PAUSE; // 将任务更改为暂停状态状态
        // 清除任务节点信息
        jd_task->node.next = JD_NULL;
        jd_task->node.previous = JD_NULL;
        return JD_OK;
    }
    /**
     * @description: 删除任务
     * @param {jd_task_t} *jd_task 任务节点指针
     * @return {*}
     */
    jd_int32_t jd_task_delete(jd_task_t *jd_task){
        if (jd_task == JD_NULL)
```

```
                return JD_ERR;

        if (jd_task == jd_task_frist)
                return JD_ERR; // 判断是否为系统第一个任务，系统第一个任务不可删除

        jd_task_pause(jd_task); // 将任务修改为暂停状态，目的是从就绪或延时链表中删除节
点
        free((jd_uint32_t *)jd_task); // 释放任务栈内存

        return JD_OK;
}
/**
 * @description: 设置任务执行完成后自动回收内存，任务销毁
 * @param {jd_task_t} *jd_task 任务节点指针
 * @return {*}
 */jd_int32_t jd_task_auto_delate(jd_task_t *jd_task){
        if (jd_task == JD_NULL)
                return JD_ERR;
        jd_task->auto_delate = JD_TASK_DELATE;
        return JD_OK;
}
```

7.9 修改异常处理函数

我们对SysTick异常处理函数进行修改以满足新设计的要求，在SysTick异常处理函数中，主要是适配新的全局变量和链表。

```
/**
 * @description: hal库已自动使能systick，以下为hal库systick异常回调函数
 * @return {*}
 */
 void HAL_IncTick(void){
        uwTick += uwTickFreq; // 系统自带不可删除,否则hal_delay等hal库函数不可用

        jd_time++; // jd_lck++
        // 判断延时表头是否到达时间，若没有到达时间，则切换，若到达时间则将任务加入到就绪
任务,再切换任务
        jd_task_t *jd_task;
        jd_task = (jd_task_t *)jd_task_list_delaying; // 获取任务数据

        while (jd_task->timeout == jd_time)
        {
```

```
                // 如果循环定时器任务，将下一次定时时间写入任务信息
                if (jd_task->timer_loop == JD_TIMER_LOOP)
                {
                        jd_task->timeout = jd_time + jd_task->timer_loop_timeout;
                }

                jd_task_list_delaying = jd_node_delete(jd_task_list_delaying,
jd_task_list_delaying); // 删除延时完成的节点

                jd_task->status = JD_TASK_READY; // 将任务更改为就绪状态
                // 加入就绪链表
                jd_task_list_readying = jd_node_in_rd(jd_task_list_readying,
&jd_task->node);

                if (jd_task_list_delaying == JD_NULL)
                        break;
                jd_task = (jd_task_t *)jd_task_list_delaying; // 获取下一个任务数据
        }

        jd_asm_pendsv_putup();
    }
```

在先前的代码实现中，我们已经利用SVCall异常来主动或被动地管理任务上下文的切换。因此，我们需要移除原有的jd_task_switch函数，在PendSV异常处理程序中，并未涵盖对任务其他数据的处理，所以我们必须对PendSV异常处理程序进行修改，将jd_task_switch函数的相关内容迁移到新的PendSV异常处理函数jd_PendSV_Handler中。

```
    /**
    * @description: PendSV处理函数
    * @return {*}
    */void jd_PendSV_Handler(void)
    {
        jd_task_t *jd_task;
        // 获取数据域
        jd_task = (jd_task_t *)jd_task_list_readying; // 获取任务数据
        jd_task_runing->status = JD_TASK_READY;
        // 任务暂停或延时状态，或者当前任务优先级低，当前任务放弃CPU使用权
        jd_task->status = JD_TASK_RUNNING;     // 即将运行的任务改为正在运行状态
        jd_task_stack_sp = &jd_task_runing->stack_sp;     // 更新当前任务全局栈指针变
量
        jd_task_runing = jd_task;      // 更改当前为运行的任务
        jd_task_next_stack_sp = &jd_task_runing->stack_sp; // 更新下一个任务全局栈指
针变量
```

```
        jd_asm_pendsv_handler(); // 切换上下文
    }
```

将新定义的jd_PendSV_Handler函数与PendSV异常处理函数绑定，移除原来PendSV异常处理函数中的调用方法。

7.10 修改任务延时函数

任务延时涉及主动切换任务的操作机制，我们已淘汰旧有的上下文切换函数，所以需要将SVCall调用集成至其中。

```
    /**
    * @description: jdos延时，让出CPU使用权
    * @param {jd_uint32_t} ms  延时时间，单位ms
    * @return {*}
    */
    void jd_delay(jd_uint32_t ms)
    {
        if (ms == 0)
        return;
        jd_asm_cps_disable();
        jd_task_runing->status = JD_TASK_DELAY;
        jd_task_runing->timeout = jd_time + ms; // 将延时时间写入节点
        // 删除就绪链表中的节点
        jd_task_list_readying = jd_node_delete(jd_task_list_readying,
&jd_task_runing->node);
        // 将节点加入延时链表
        jd_task_list_delaying = jd_node_in_rd(jd_task_list_delaying,
&jd_task_runing->node);
        // 切换线程，让出CPU，等延时后调度，用svc指令
        jd_task_t *jd_task;
        // 获取数据域
        jd_task = (jd_task_t *)jd_task_list_readying; // 获取任务数据
        // 任务暂停或延时状态，或者当前任务优先级低，当前任务放弃CPU使用权
        jd_task->status = JD_TASK_RUNNING;      // 即将运行的任务改为正在运行状态
        jd_task_stack_sp = &jd_task_runing->stack_sp;    // 更新当前任务全局栈指针变
量
        jd_task_runing = jd_task;      // 更改当前为运行的任务
        jd_task_next_stack_sp = &jd_task_runing->stack_sp; // 更新下一个任务全局栈指
针变量
        jd_asm_svc_task_switch();
    }
```

7.11 其他修改

在jdos.s文件中引入必要的变量。

```
IMPORT jd_task_entry
IMPORT jd_task_exit_entry
```

修改任务就绪函数。

```
/**
* @description: 将任务加入就绪链表
* @param {jd_task_t} *jd_task 任务节点指针
* @return {*}
*/
jd_int32_t jd_task_run(jd_task_t *jd_task)
{
      if (jd_task == JD_NULL)
            return JD_ERR;
      jd_task->status = JD_TASK_READY; // 将任务更改为就绪状态
      // 加入就绪链表
      jd_task_list_readying = jd_node_in_rd(jd_task_list_readying,
&jd_task->node);
      // 切换任务
      jd_asm_pendsv_putup();
      // 插入节点
      return JD_OK;
}
```

修改初始化函数。

```
/**
* @description: jd初始化
* @return {*}
*/jd_int32_t jd_init(void)
{
      // 初始化链表
      jd_task_list_readying = JD_NULL;
      jd_task_list_delaying = JD_NULL;
      // 设置优先级为最低
      jd_task_frist = jd_task_create(jd_main, JD_DEFAULT_STACK_SIZE, 127);
      while (jd_task_frist == JD_NULL); // 空闲任务不能创建则死循环
      all_register_t *stack_register = (struct all_register
*)jd_task_frist->stack_sp; // 将指针转换成寄存器指针
      // jd_main任务没有退出的程序，故返回地址指向自己
      stack_register->lr = (jd_uint32_t)jd_main;
      jd_task_frist->status = JD_TASK_READY; // 任务就绪
```

```
        // jd_task_frist->node->addr = jd_task_frist; // 记录节点内存地址，方便通过节点找到任务数据域
        jd_task_list_readying = &jd_task_frist->node; // 将任务挂在就绪链表上
        jd_task_runing = jd_task_frist;                          // 保存当前任务为正在运行任务
        // jd_asm_systick_init(); //启动systick,hal库已自动使能systick
        // 进入空闲任务
        jd_asm_task_first_switch(&jd_task_frist->stack_sp, jd_main);
        return JD_OK;
    }
```

在jd_main中创建任务时增加优先级。

```
    jd_task_t *test_task1, *test_task2, *test_task3;/**
    * @description: 系统main,系统第一个任务，不可使用jd_task_delete删除，可添加其他任务初始化代码
    * @return {*}
    */
    __weak void jd_main(void){
        test_task1 = jd_task_create(task1, 512, 3);
        if (test_task1 != JD_NULL)
                jd_task_run(test_task1);
        test_task2 = jd_task_create(task2, 512, 1);
        if (test_task1 != JD_NULL)
                jd_task_run(test_task2);
        test_task3 = jd_task_create(task3, 512, 2);
        if (test_task1 != JD_NULL)
                jd_task_run(test_task3);
        while (1)
        {
        // 注意此处调用延时切换任务，如果所有任务都不为就绪状态，程序将死循环，直到有就绪任务才会切换
        // 应该在此处休眠或者其他不重要的工作
        };
    }
```

我们还需要对头文件中的相关声明进行必要的修改，直到能够成功编译代码。

```
    /******************全局变量***********************/
    extern jd_node_list_t *jd_task_list_readying; // 创建就绪任务链表
    extern jd_node_list_t *jd_task_list_delaying; // 创建延时任务链表
    extern jd_task_t *jd_task_runing;           // 创建当前任务指针
    extern void *jd_task_stack_sp;              // 创建当前任务堆栈指针的地址
    extern void *jd_task_next_stack_sp;         // 创建下一个任务堆栈指针的地址
    extern jd_task_t *jd_task_frist;            // 创建一个系统空闲任务
    extern jd_uint32_t jd_task_entry;           // 任务入口
    extern jd_uint32_t jd_task_exit_entry;      // 任务exit入口
```

```
    extern jd_time_t jd_time;                   // 系统时钟，单位ms
    /*****************汇编函数***********************/
    extern void jd_asm_task_first_switch(jd_uint32_t *, void *); // 第一次进入任务
    extern void jd_asm_pendsv_putup(void);                // 切换任务节点，悬挂
PendSV异常，PendSV中进行上下文切换
    extern void jd_asm_pendsv_handler(void);              // PendSV切换上下文
    extern void jd_asm_systick_init(void);                // systick初始化
    extern void jd_asm_cps_disable(void);                 // 除能 NMI 和硬 fault
之外的所有异常
    extern void jd_asm_cps_enable(void);                  // 使能中断
    extern void jd_asm_svc_handler(void);                 // svc异常处理
    extern void jd_asm_svc_task_switch(void);             // 任务上下文切换
    extern void jd_asm_svc_task_exit(void);               // 任务退出
    /*****************jd_timer***********************/
    void jd_delay(jd_uint32_t ms);
    /*****************jd_task***********************/
    jd_task_t *jd_task_create(void (*task_entry)(), jd_uint32_t stack_size, jd_int8_t
priority); // 创建任务
    jd_int32_t jd_task_delete(jd_task_t *jd_task); // 删除任务
    jd_int32_t jd_task_auto_delate(jd_task_t *jd_task); // 设置任务运行完成后自动回收内
存，删除任务
    jd_int32_t jd_task_run(jd_task_t *jd_task); // 将任务加入就绪链表
    jd_int32_t jd_task_pause(jd_task_t *jd_task); // 任务暂停
    jd_int32_t jd_init(void); // jd初始化
    void jd_main(void); // jd main
    jd_int32_t jd_node_insert(jd_node_list_t *node_previous, jd_node_list_t *node,
jd_node_list_t *node_next); // 节点连接函数
    jd_node_list_t *jd_node_delete(jd_node_list_t *list, jd_node_list_t *node); // 删
除节点
    jd_int64_t compare_priority(jd_task_t *task1, jd_task_t *task2); // 比较函数，用于
jd_node_in_rd中使用
    jd_node_list_t *jd_node_in_rd(jd_node_list_t *list, jd_node_list_t *node); // 将
节点插入就绪或者延时链表
    void jd_task_exit(void); // 任务执行完成后由系统调用
```

下载编译好的代码，我们注意到LED灯开始快速且无规则地闪烁。通过Keil调试，我们为测试任务 1、2 和 3 设置断点，观察到任务的执行顺序与我们设定的优先级相匹配，这验证了我们成功地实现了任务抢占式功能。

7.12 文件分离

在我们的项目中，jdos.c文件已经变得非常庞大，包含了大量的代码。为了

提高代码的可维护性和可读性，我们决定采取一些措施，将现有的代码分散到多个新的文件中。这样做不仅可以使每个文件的职责更加明确，还能方便我们在未来进行代码的修改和扩展。通过创建更多的文件，我们希望能够更好地组织和管理我们的代码库，从而提高整体的开发效率和项目的可维护性。

我们将文件名jdos.c修改为jd_task.c，jdos.s更改为jd_cm3.s，并新创建jd_it.c、jd_main.c以及jd_timer.c文件。与任务数据处理相关的函数现位于jd_task.c内，异常处理函数则被归类于jd_it.c，jd_main.c中包含了空闲函数，而与时间管理相关的函数则被放置在jd_timer.c中。

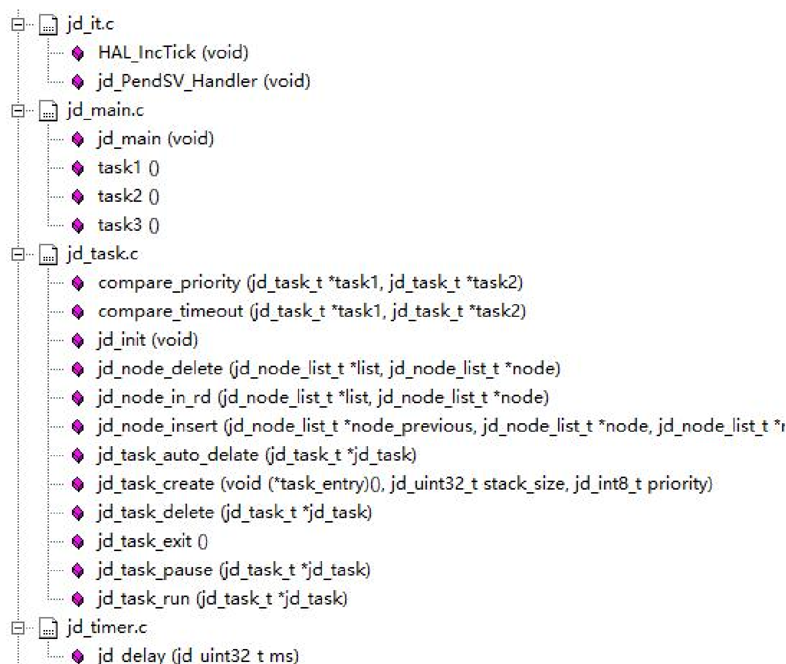

图 45 函数对应文件图

小结

在这一章节中，我们主要完成了任务抢占式功能的开发和实现，重新设计了任务管理流程，使得系统能够根据任务的优先级调整任务的执行顺序，通过这种方式，高优先级的任务可以打断低优先级任务的执行，从而确保关键任务能够迅速得到处理。

第八章 定时任务

8.1 定时任务的特点

有时，我们需要周期性地执行特定任务，或者在延时后才能运行的任务，这类任务统称为定时任务。

定时任务是任务管理功能的延伸，它使得系统能够按照预定的时间间隔自动执行特定任务。为了实现这一功能，我们将定时任务纳入延时链表的管理范畴，在这个过程中，定时任务与延时任务本质上是相同的。

8.2 定时任务函数编写

在jd_timer.c文件中，我们添加用于启动和停止定时任务的函数。启动定时任务是将一个任务设置为定时执行，并根据定时器的状态（循环或非循环）来决定任务的执行方式，它涉及到任务调度器中的链表操作，包括从就绪链表中移除任务和将其加入到延时链表中。停止定时任务是停止一个任务的定时设置，使其不再按照之前设置的定时条件执行，将任务从链表中删除，任务将不再受到定时控制。

```
/**
 * @description: 定时任务创建
 * @param {jd_task_t} *jd_task 创建的普通任务
 * @param {jd_uint32_t} ms 定时时间
 * @param {jd_timer_status_t} timer_status 是否为循环任务
 * @return {*}
 */
jd_int32_t jd_timer_start(jd_task_t *jd_task, jd_uint32_t ms, jd_timer_status_t timer_status)
{
	if (jd_task == JD_NULL)
		return JD_ERR;
	// 定时任务，不是循环
	if (timer_status == JD_TIMER_NOLOOP)
	{
		jd_task->timer_loop = JD_TIMER_NOLOOP;
	}
```

```
        // 循环定时任务
        else if (timer_status == JD_TIMER_LOOP)
        {
                jd_task->timer_loop = JD_TIMER_LOOP;
        }
        else
        {
        return JD_ERR;
        }
        // 定时时间
        jd_task->timer_loop_timeout = ms;
        // 将延时时间写入节点
        jd_task->timeout = jd_time + jd_task->timer_loop_timeout;
        // 定义寄存器
        all_register_t *stack_register = (struct all_register *)jd_task->stack_sp;
        // 定时任务执行完成，执行销毁程序
        stack_register->lr = (jd_uint32_t)jd_task_exit;
        // 判断是否在就绪链表中
        if (jd_task->status == JD_TASK_RUNNING || jd_task->status == JD_TASK_READY)
        {
                // 删除就绪链表中的节点
                jd_task_list_readying = jd_node_delete(jd_task_list_readying,
&jd_task->node);
        }
        // 将节点加入延时链表
        jd_task_list_delaying = jd_node_in_rd(jd_task_list_delaying,
&jd_task->node);
        return JD_OK;
    }/*timer删除
    * jd_task:创建的普通任务
    */
    jd_int32_t jd_timer_stop(jd_task_t *jd_task)
    {
        if (jd_task == JD_NULL)
                return JD_ERR;
        // 暂停任务，将任务从链表中删除
        jd_task_pause(jd_task);
        // 关闭定时器任
        jd_task->timer_loop = JD_TIMER_NOTIMER;
        return JD_OK;
    }
```

小结

本章实现了对任务定时执行功能，增强了任务管理的灵活性和效率。

第九章 内存管理

9.1 内存管理设计

在我们之前的代码实现中，缺乏自主的内存管理机制，依赖于C语言标准库提供的malloc和free函数来处理任务的内存分配与释放。然而，在嵌入式系统中，标准库函数的使用存在若干问题，例如不确定的内存分配时间、频繁的分配与释放操作导致的内存碎片化，以及受限于系统堆大小的局限性等。这些问题对于实时操作系统而言，可能会造成严重的性能影响。因此，我们需要设计一套定制的内存管理算法，以增强系统的整体稳定性。

我们将划分一大块内存进行管理，姑且就把这一块内存叫做内存池，当任务正在被创建时或者用户申请使用，系统将会在内存池中寻找一块大小合适的内存进行划分，当内存池中没有足够的大小的内存时，将申请失败。

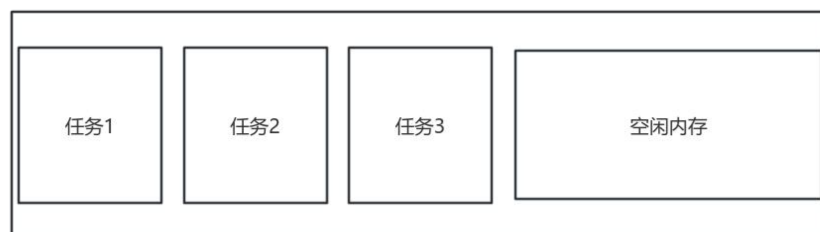

图 46 内存使用分布

在手动释放内存的过程中，系统会回收被删除任务占用的内存空间。若释放的内存区域周围存在未使用的内存空间，这些内存将会被合并，形成一块较大的连续内存区域，从而有效减少内存碎片化现象。

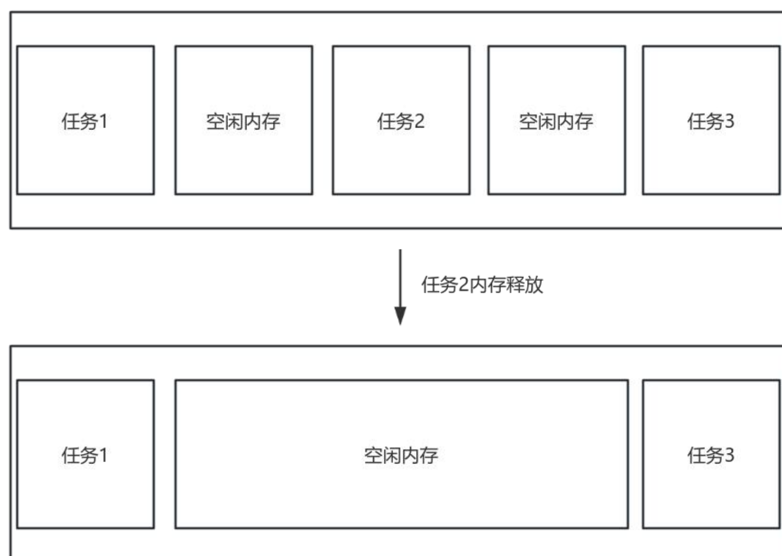

图 47 内存合并示意

9.2 内存申请

我们定义一个内存控制块，其中包含内存的链表、内存使用状态和当前内存块大小信息，同时宏定义一个 MEM_MAX_SIZE用于表示内存池最大的空间大小。

```
/*内存使用状态*/
typedef enum jd_mem_used
{
      JD_MEM_USED = 1,
      JD_MEM_FREE,
} jd_mem_used_t;
#pragma pack(4) // 4 字节对齐/*内存控制块*/
typedef struct jd_mem
{
      jd_node_list_t node;  // 链表节点
      jd_mem_used_t used;   // 当前内存是否被使用
      jd_uint32_t mem_size; // 当前整体内存块大小
} jd_mem_t;
#pragma pack() // 取消结构体对齐
/*开辟内存大小*/
#define MEM_MAX_SIZE (1024 * 8)
```

创建一个新文件jd_memory.c，用于存放内存管理算法，我们模仿标准库的分配函数名称，创建一个jd_malloc分配函数。

```
    jd_mem_t *jd_mem_use = JD_NULL; // （1）jd_uint8_t jd_mem_space[MEM_MAX_SIZE];// （2）
/**
    * @description: 分配内存空间
    * @param {jd_uint32_t} mem_size 需要分配的空间
    * @return {*}
    */void *jd_malloc(jd_uint32_t mem_size){
        jd_mem_t *jd_mem_temp, *jd_mem_new_free;// （3）
        jd_mem_temp = jd_mem_use;// （4）
        while (1) // （5）
        {
                // 找到足够的空闲空间
                if (jd_mem_temp->used == JD_MEM_FREE && (mem_size + sizeof(jd_mem_t)
<= jd_mem_temp->mem_size))// （6）
                {
                        jd_mem_temp->used = JD_MEM_USED; // 标记为使用状态（7）
                        // 防止内存管理中出现泄露,剩余内存足够分割至少
sizeof(jd_mem_t)+1 的空间
                        if ((jd_mem_temp->mem_size - mem_size) >
sizeof(jd_mem_t))// （8）
                        {
                                jd_mem_new_free = (jd_mem_t *)(((jd_uint8_t
*)jd_mem_temp) + mem_size + sizeof(jd_mem_t)); // 将剩余的内存添加上内存块信息 （9）
                                jd_mem_new_free->mem_size =
jd_mem_temp->mem_size - mem_size - sizeof(jd_mem_t);         // 剩余内存大小 （10）
                                jd_mem_new_free->used = JD_MEM_FREE; // 标记为空
闲内存（11）
                                jd_mem_temp->mem_size = mem_size +
sizeof(jd_mem_t); // 标记当前内存块总大小 （12）
                                // 下一个控制块存在
                                if (jd_mem_temp->node.next != JD_NULL) // （13）
                                {
                                        jd_mem_t *jd_mem_new_next;
                                        jd_mem_new_next = (jd_mem_t
*)jd_mem_temp->node.next;
                                        jd_node_insert(&jd_mem_temp->node,
&jd_mem_new_free->node, &jd_mem_new_next->node); // 插入内存节点
                                }
                                // 下一个不存在
                                else  // （14）
                                {
                                        jd_node_insert(&jd_mem_temp->node,
&jd_mem_new_free->node, JD_NULL); // 插入内存节点
                                }
                        }
```

```
                    break; // （15）
                }
                // 遍历完成，没有足够的空间进行分配，返回JD_NULL
                if (jd_mem_temp->node.next == JD_NULL)
                {
                    return JD_NULL; // （16）
                }
                jd_mem_temp = (jd_mem_t *)jd_mem_temp->node.next; // （17）
        }
        return (void *)(((jd_uint8_t *)jd_mem_temp) + sizeof(jd_mem_t)); // 返回分配的地址 （18）
    }
```

上述代码中（1）定义全局内存链表；

（2）定义一块大内存用于内存管理；

（3）定义两个jd_mem_t类型的指针变量jd_mem_temp和jd_mem_new_free，分别用于临时存储当前遍历的内存块和新分配的空闲内存块；

（4）将jd_mem_temp初始化为指向jd_mem_use，指向当前可用的内存块链表的头部；

（5）遍历可用内存链表；

（6）找到足够的空闲空间，用于可以分配内存；

（7）将找到的内存标记为使用状态；

（8）如果剩余的内存大于一个内存控制块的大小，则可以分割出一个新的空闲内存块；

（9）给剩余的空闲内存块添加内存信息；

（10）设置空闲剩余内存大小；

（11）标记为空闲内存；

（12）设置当前申请的内存大小；

（13）检查当前内存块 jd_mem_temp 是否有下一个内存块，有下一个内存块则将申请内存插入到表中；

（14）检查当前内存块 jd_mem_temp 是否有下一个内存块，没有下一个内存块则将申请内存插入到表尾；

（15）分配成功后，跳出遍历循环；

（16）如果遍历完所有内存块后没有找到足够大的空闲块，返回 JD_NULL

表示分配失败。

（17）如果当前内存块不是最后一个，更新 jd_mem_temp 为下一个内存块的地址，继续遍历；

（18）返回分配的内存地址，地址是当前内存块的起始地址加上 sizeof(jd_mem_t)，即跳过内存块头部信息，返回实际可用的内存空间地址。

9.3 内存释放

创建一个内存释放函数jd_free，释放jd_malloc申请的空间，同时在函数中进行内存合并操作。

```
/**
 * @description: 释放内存空间
 * @param {void} *ptr 传入申请的空间的地址
 * @return {*}
 */
void jd_free(void *ptr)
{
    jd_mem_t *jd_mem_old, *jd_mem_previous, *jd_mem_next, *jd_mem_next_next;
    jd_mem_old = (jd_mem_t *)((jd_uint8_t *)ptr - sizeof(jd_mem_t)); // 获取控制块
信息 （1）

    jd_mem_old->used = JD_MEM_FREE; //（2）
    // 下一个控制块存在
    if (jd_mem_old->node.next != JD_NULL)//（3）
    {
        jd_mem_next = (jd_mem_t *)jd_mem_old->node.next;//（4）
        // 判断下一个内存块是free
        if (jd_mem_next->used == JD_MEM_FREE)//（5）
        {
            // 合并内存块
            jd_mem_old->mem_size += jd_mem_next->mem_size;//（6）

            // 判断下一个控制块 的下一个存在
            if (jd_mem_next->node.next != JD_NULL)//（7）
            {
                jd_mem_next_next = (jd_mem_t *)jd_mem_next->node.next;
                jd_node_insert(&jd_mem_old->node, JD_NULL,
&jd_mem_next_next->node);
            }
            // 不存在
```

```
                else //（8）
                {
                    jd_mem_old->node.next = JD_NULL;
                }
            }
        }

        // 上一个控制块存在
        if (jd_mem_old->node.previous != JD_NULL)//（9）
        {
            // 获得上一个控制块的信息
            jd_mem_previous = (jd_mem_t *)jd_mem_old->node.previous;//（10）

            // 判断上一个内存块是free
            if (jd_mem_previous->used == JD_MEM_FREE)//（11）
            {
                jd_mem_previous->mem_size += jd_mem_old->mem_size; // 合并内存块（12）
                // 判断下一块内存 存在
                if (jd_mem_old->node.next != JD_NULL) //（13）
                {
                    jd_mem_next = (jd_mem_t *)jd_mem_old->node.next;
                    jd_node_insert(&jd_mem_previous->node, JD_NULL,
&jd_mem_next->node);
                }
                // 下一块内存不存在
                else//（14）
                {
                    jd_mem_previous->node.next = JD_NULL;
                }
            }
        }
        ptr = JD_NULL;//（15）
    }
```

上述代码中（1）通过传入的指针ptr减去sizeof(jd_mem_t)来获取对应的内存块的控制块信息，这是因为在jd_malloc中，实际返回给用户的内存地址是控制块之后的位置，所以需要回退到控制块；

（2）将要释放的内存标记为空闲状态；

（3）检查当前内存块的下一个控制块是否存在；

（4）如果存在，获取下一个控制块的指针；

（5）检查下一个控制块是否也是空闲的；

（6）如果下一个控制块是空闲的，将当前内存块的大小 mem_size 增加下一个控制块的大小，实现内存块的合并；

（7）检查下一个控制块的下一个控制块是否存在，存在则将合并后的内存插入到表中；

（8）不存在则将合并后的内存插入到表尾；

（9）检查当前内存块的上一个控制块是否存在；

（10）如果存在，获取上一个控制块的指针；

（11）检查上一个控制块是否也是空闲的；

（12）如果上一个控制块是空闲的，将上一个控制块的大小 mem_size 增加当前内存块的大小，实现内存块的合并；

（13）检查当前内存块的下一个控制块是否存在，存在则插入到表中；

（14）不存在则插入到表尾；

（15）将传入的指针 ptr 设置为 JD_NULL，这是一个防御性编程的做法，以防止后续代码错误地使用已经释放的内存地址。

9.4 内存初始化

我们对内存进行初始化，设置一个初始的、未分配的内存池，以便后续可以通过 jd_malloc 和 jd_free 进行内存分配和释放。

```
/**
 * @description: 内存初始化
 * @return {*}
 */
jd_uint32_t jd_mem_init()
{
    jd_mem_use = (jd_mem_t *)jd_mem_space; // 传入内存块地址
    // jd_mem_use->node.addr = jd_mem_use;    // 保存内存块地址

    jd_mem_use->node.next = JD_NULL;
    jd_mem_use->node.previous = JD_NULL;

    jd_mem_use->used = JD_MEM_FREE;       // 初始为空闲内存
    jd_mem_use->mem_size = MEM_MAX_SIZE; // 初始内存块大小

    return JD_OK;
```

```
}
```

在jd_init函数中调用以上初始化函数，同时在jd_task_create中将mallo函数更改为jd_malloc，在jd_task_delete中将free更改为jd_free，别忘了jd_malloc和jd_free在jdos.h中声明。

我们在jdos.h中取消keil的标准库的引用，同时在Keil设置中取消勾选Use MicroLIB。

9.5 系统栈与任务栈分离

我们将系统栈与用户栈进行分离，以确保系统运行的稳定性和任务执行的独立性。系统栈主要负责内核操作和异常处理，而用户栈则用于执行应用程序代码。通过这种方式，可以有效避免因用户任务错误操作导致的系统崩溃，同时简化了任务切换过程，提高了系统的响应速度和效率。

利用以下汇编代码，我们可以使任务程序使用进程堆栈指针PSP，而主堆栈指针MSP则是系统进入异常处理程序是自动切换，因此我们只需要在任务上下文切换时，更改为进程堆栈指针即可。

```
MOV R0, #0x2 ; 设置CONTROL寄存器，让用户程序使用PSP
MSR CONTROL,R0
```

需要更改的汇编函数有jd_asm_task_first_switch、jd_asm_pendsv_handler和jd_asm_task_exit_switch，在函数最后中断响应开启前添加以上代码。

小结

本章主要实现了自定义的内存管理机制，通过自定义的方式管理内存，确保了内存分配和释放的实现。我们通过jd_malloc和jd_free函数来实现内存的动态分配和回收，同时通过合并相邻的空闲内存块来减少内存碎片，提高内存利用率。

在系统栈与任务栈分离的设计中，我们通过汇编指令调整了任务的堆栈指针，确保了系统栈和任务栈的独立性。

我们在Keil工程中取消标准库stdio.h的引用，并在Keil设置中取消勾选Use MicroLIB，以避免与自定义内存管理函数发生冲突。

第十章 Hello world！实现

Hello world！任何程序的开始，无论你是学习什么语言，都是从这个简单的示例开始。这个示例程序通常会在屏幕打印出“Hello, World!”，从而帮助初学者理解基本的语法和程序结构。在嵌入式系统编程中，这个示例同样重要，因为它帮助开发者熟悉硬件和软件的交互方式，接下来，让我们在jdos中实现这一示例。

10.1 串口配置

我们在STM32CubeMX中对芯片的串口进行配置，设置一些串口的信息，比如波特率、数据位、停止位和校验位等参数。

打开工程目录下.ioc后缀的文件，在这个熟悉的界面中按照以下图片中的步骤进行设置，最后重新生成代码。

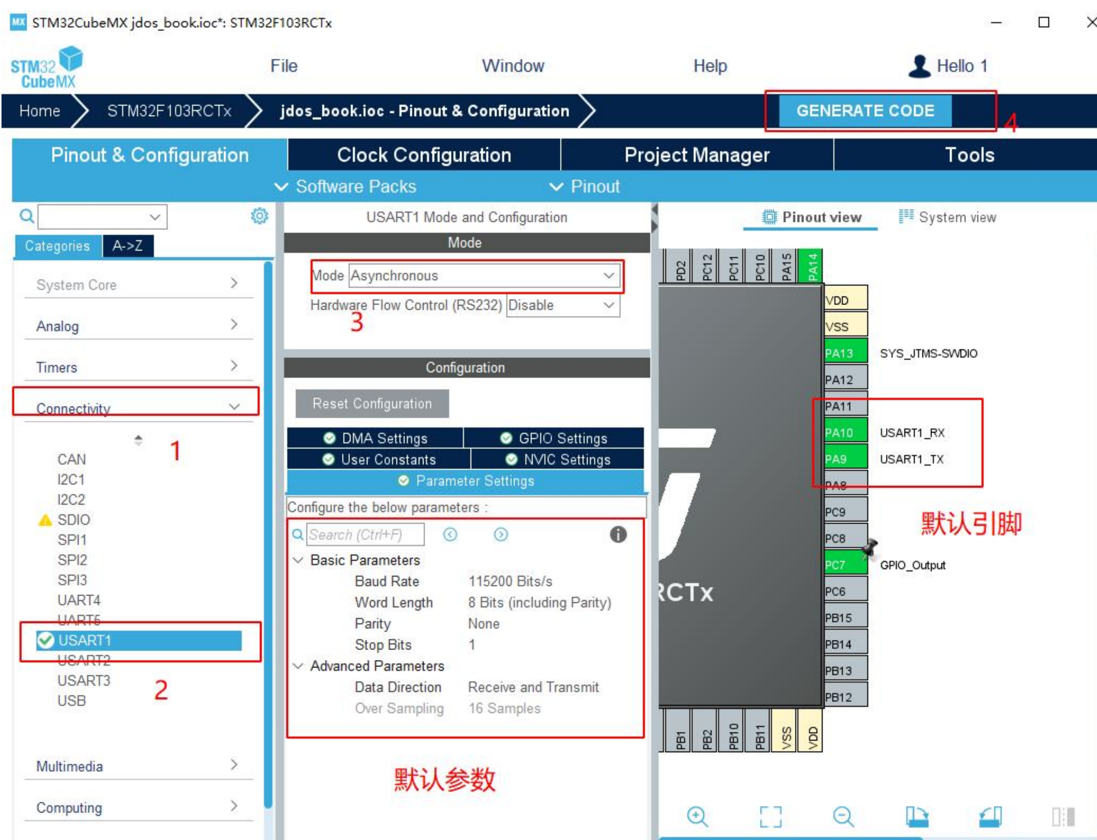

图 48 串口配置

打开Keil工程，重新编译以下代码，编译通过，HAL库已经帮组我们初始化好了串口。

10.2 打印实现

创建一个名为jd_printf.c的新文件，用于实现我们的打印功能代码，我们先实现底层硬件的对接，jd_transmit是串口对接函数，里面包含HAL库的串口发送函数，这个函数直接与底层进行通信，jd_putjd_int8_t是发送单个字符，jd_printstring是发送字符串。

```
#include "jdos.h"
#include <stdarg.h> // 包含标准变长参数库
// 外部定义的UART句柄
extern UART_HandleTypeDef huart1;
/**
 * @description: 通过UART发送数据
 * @param {jd_uint8_t*} pData 数据指针
 * @param {jd_uint16_t} Size 数据大小
 * @return {*}
 */void jd_transmit(jd_uint8_t *pData, jd_uint16_t Size){
    // 此处对接硬件发送的接口
    HAL_UART_Transmit(&huart1, pData, Size, HAL_MAX_DELAY);
}
/**
 * @description:  发送单个字符
 * @param {jd_int8_t} ch 要发送的字符
 * @return {*}
 */void jd_putjd_int8_t(jd_int8_t ch){
    jd_transmit((jd_uint8_t *)&ch, 1);
}
/**
 * @description:  发送字符串
 * @param {jd_int8_t*} str 要发送的字符串
 * @return {*}
 */void jd_printstring(const jd_int8_t *str){
    while (*str)
    {
        jd_putjd_int8_t(*str++);
    }
}
```

在实现完整的打印函数之前，我们还需要对整数和小数的打印进行处理，

jd_printint为打印整数，jd_printfloat为打印小数。

```
   /**
    * @description:  打印整数
    * @param {jd_int32_t} num 要打印的整数
    * @param {jd_int32_t} base 进制（如10 进制、16 进制等）
    * @param {jd_int32_t} width 最小宽度
    * @param {jd_int32_t} zero_pad 是否用0 填充
    * @return {*}
    */void jd_printint(jd_int32_t num, jd_int32_t base, jd_int32_t width, jd_int32_t
zero_pad){
       jd_int8_t buffer[32]; // 足够存储整数和符号
       jd_int32_t i = 0;
       jd_int32_t is_negative = 0;

       if (num < 0)
       {
           is_negative = 1;
           num = -num;
       }

       do
       {
           jd_int32_t digit = num % base;
           buffer[i++] = (digit < 10) ? (digit + '0') : (digit - 10 + 'A');
           num /= base;
       } while (num > 0);

       if (is_negative)
       {
           buffer[i++] = '-';
       }

       while (i < width)
       {
           buffer[i++] = zero_pad ? '0' : ' ';
       }

       for (jd_int32_t j = i - 1; j >= 0; j--)
       {
           jd_putjd_int8_t(buffer[j]);
       }
   }
   /**
    * @description: 打印浮点数
```

```
 * @param {float} num  要打印的浮点数
 * @param {jd_int32_t} precision 小数点后的精度
 * @param {jd_int32_t} width 最小宽度
 * @return {*}
 */void jd_printfloat(float num, jd_int32_t precision, jd_int32_t width){
    if (num < 0)
    {
        jd_putjd_int8_t('-');
        num = -num;
    }

    jd_int32_t int_part = (jd_int32_t)num;
    float frac_part = num - int_part;

    jd_printint(int_part, 10, width - precision - 1, 0);
    jd_putjd_int8_t('.');

    for (jd_int32_t i = 0; i < precision; i++)
    {
        frac_part *= 10;
        jd_int32_t digit = (jd_int32_t)frac_part;
        jd_putjd_int8_t(digit + '0');
        frac_part -= digit;
    }
}
```

我们已经在底层对接了硬件串口，处理了打印字符串、小数和整数，接下来我们实现jd_printf函数，它接收一个可变长度的参数，用法与标准库的printf一致，在jd_printf中我们对接收到的数据进行判断，不同是数据选择不同的打印方法，从而实现打印功能。

```
/**
 * @description:  格式化打印函数
 * @param {jd_int8_t*} format  格式化字符串
 * @param ... 可变参数列表
 * @return {*}
 */void jd_printf(const jd_int8_t *format, ...){
    va_list args;
    va_start(args, format);

    while (*format)
    {
        if (*format == '%')
        {
```

```
format++;
jd_int32_t width = 0;
jd_int32_t precision = 6; // 默认精度为6 位小数
jd_int32_t zero_pad = 0;

// 解析宽度
while (*format >= '0' && *format <= '9')
{
    width = width * 10 + (*format - '0');
    format++;
}

// 解析精度
if (*format == '.')
{
    format++;
    precision = 0;
    while (*format >= '0' && *format <= '9')
    {
        precision = precision * 10 + (*format - '0');
        format++;
    }
}

// 解析对齐和填充
if (*format == '0')
{
    zero_pad = 1;
    format++;
}

switch (*format)
{
case 'c':
{
    jd_int8_t ch = (jd_int8_t)va_arg(args, jd_int32_t);
    jd_putjd_int8_t(ch);
    break;
}
case 'd':
{
    jd_int32_t num = va_arg(args, jd_int32_t);
    jd_printint(num, 10, width, zero_pad);
    break;
```

```
            }
            case 'f':
            {
                float num = (float)va_arg(args, double);
                jd_printfloat(num, precision, width);
                break;
            }
            case 's':
            {
                const jd_int8_t *str = va_arg(args, const jd_int8_t *);
                jd_printstring(str);
                break;
            }
            case 'x':
            {
                jd_int32_t num = va_arg(args, jd_int32_t);
                jd_printint(num, 16, width, zero_pad);
                break;
            }
            case 'X':
            {
                jd_int32_t num = va_arg(args, jd_int32_t);
                jd_printint(num, 16, width, zero_pad);
                break;
            }
            case 'p':
            {
                void *ptr = va_arg(args, void *);
                jd_printint((jd_uint32_t)ptr, 16, width, zero_pad);
                break;
            }
            case 'u':
            {
                jd_uint32_t num = va_arg(args, jd_uint32_t);
                jd_printint(num, 10, width, zero_pad);
                break;
            }
            case 'o':
            {
                jd_uint32_t num = va_arg(args, jd_uint32_t);
                jd_printint(num, 8, width, zero_pad);
                break;
            }
            case 'b':
```

```
            {
                jd_uint32_t num = va_arg(args, jd_uint32_t);
                jd_printint(num, 2, width, zero_pad);
                break;
            }
            case '%':
            {
                jd_putjd_int8_t('%');
                break;
            }
            default:
                jd_putjd_int8_t(*format);
                break;
            }
        }
        else
        {
            jd_putjd_int8_t(*format);
        }
        format++;
    }

    va_end(args);
}
```

我们在jd_main函数中添加一些打印测试。

```
jd_printf("Hello world!\r\n");
```

并在任务中也添加相应的打印测试，将单片机的串口与CH340 芯片连接，再将CH340 芯片与电脑连接。请确保CH340 的驱动程序已正确安装。随后，打开串口通信软件，接下来是测试结果的展示。

```
Hello world!
Hello test3
Hello test1
Hello test2
Hello test3
Hello test1
Hello test3
Hello test2
Hello test1
Hello test3
Hello test1
Hello test2
Hello test3
Hello test1
Hello test3
```

图 49 打印测试结果

小结

本章通过配置串口和编写相应的打印函数，我们成功地输出了"Hello World!"，这一过程验证我们的硬件连接和软件配置的正确性。

第十一章 内核休眠

11.1 内核休眠原理

内核休眠功能允许系统在没有任务需要执行时进入低功耗模式。在内核休眠状态下，处理器的时钟可以被关闭或降低频率，以减少能量消耗。当有新的任务就绪或外部中断发生时，系统将被唤醒，恢复正常的运行频率和任务执行。实现内核休眠功能需要对中断管理、任务调度和时钟控制有深入的理解和精确的控制。通过合理设计，可以有效延长设备的电池寿命，尤其适用于便携式或电池供电的嵌入式系统。

在Cortex-M3 中，提供了两种睡眠模式，它们由系统控制寄存器的值来确定。

表 5 系统状态寄存器部分位（0xE000ED10）

位段	名称	复位值	描述
2	SLEEPDEEP	0	当进入睡眠模式时，使能外部的SLEEPDEEP信号，以允许停止系统时钟
1	SLEEPONEXIT	-	激活“SleepOnExit”功能

SleepOnExit功能激活后，处理器在完成当前异常处理后，如果返回到主程序，会自动进入睡眠模式，而不是继续执行主程序。这允许系统在异常处理完毕后，如果无其他任务需要立即执行，可以节省能源，降低功耗。

将SLEEPDEEP和SLEEPONEXIT进行排列可以得出不同的睡眠模式。

表 6 不同的睡眠模式

	SLEEPONEXIT=0	SLEEPONEXIT=1
SLEEPDEEP=0	立即睡眠模式	等待中断完成后进入睡眠模式
SLEEPDEEP=1	停止模式	

设置停止模式后，Cortex-M3 将会进入深度睡眠，为了进入停止模式，所有的外部中断的请求位(挂起寄存器(EXTI_PR))和RTC的闹钟标志都必须被清除，否则停止模式的进入流程将会被跳过，程序继续运行。

我们利用WFI或者WFE指令来开启内核的休眠功能。WFI（Wait For Interrupt）指令介绍：WFI指令是ARM架构中用于让处理器进入低功耗模式的一种指令。

当系统执行到这条指令时，如果没有任何中断发生，处理器将停止执行指令，进入等待状态，直到有中断信号唤醒它。这种机制对于延长电池寿命和降低设备能耗非常有效，特别是在不需要持续处理任务的嵌入式系统中。通过合理配置WFI指令，可以确保在没有任务执行时，系统能够自动进入低功耗模式，而在有任务需要处理时，能够迅速响应并恢复到正常工作状态。

WFE（Wait For Event）指令介绍：与WFI指令类似，WFE指令也是ARM架构中用于实现处理器低功耗模式的一种指令。不过，与WFI指令等待中断信号不同，WFE指令是等待特定的事件信号。当系统执行到WFE指令时，如果没有指定的事件信号发生，处理器同样会停止执行指令，进入低功耗的等待状态。一旦接收到指定的事件信号，处理器就会从等待状态中恢复，继续执行后续指令。WFE指令在需要等待特定外部事件触发时非常有用，能够帮助系统进一步降低能耗。

11.2 休眠实现

我们在jd_cm3.s中添加以下代码。

```
jd_asm_power_sleep        PROC
                          EXPORT jd_asm_power_sleep

                          ;SLEEPDEEP = 0;进入轻度睡眠，内核停止，外设不停止
                          ;SLEEP-NOW: 如果SLEEPONEXIT位被清除，当WRI或WFE被执行时，微

                          LDR R0,=JD_POWER_SLEEP
                          LDR R1,[R0]
                          AND R1,#0xf9 ;轻度睡眠 立即执行
                          ;ORR R1,#0x02 ;激活SleepOnExit功能，最低优先级中断执行完成后
                          ;ORR R1,#0x04 ;停止模式
                          STR R1,[R0]
                          WFI
                          ;WFE

                          BX LR
                          ENDP
```

别忘了定义系统控制寄存器的地址。

```
JD_POWER_SLEEP                EQU 0xE000ED10      ;睡眠控制寄存器
```

我们在空闲任务中引入休眠功能，以便在系统无其他任务执行时自动进入休

眠状态，有效降低功耗。

```
40 __weak void jd_main(void){
41
42   test_task1 = jd_task_create(task1, 512, 3);
43   if (test_task1 != JD_NULL)
44     jd_task_run(test_task1);
45
46   test_task2 = jd_task_create(task2, 512, 1);
47   if (test_task1 != JD_NULL)
48     jd_task_run(test_task2);
49
50   test_task3 = jd_task_create(task3, 512, 2);
51   if (test_task1 != JD_NULL)
52     jd_task_run(test_task3);
53
54   jd_printf("Hello world!\r\n");
55
56   while (1)
57   {
58     // 注意此处调用延时切换任务，如果所有任务都不为就绪状态，程序将死循环，直到有就绪任务才会切换
59     // 应该在此处休眠或者其他不重要的工作
60     jd_asm_power_sleep();
61   };
62 }
```

图 50 空闲任务休眠

第十二章 CPU使用率统计

12.1 统计原理

在一个RTOS中，我们通常需要了解CPU的使用率，以便于评估系统当前的性能状况。通过监控CPU的使用率，我们可以判断当前的任务工作量是否合理。如果CPU的使用率较低，这可能意味着任务的工作量并不饱和，系统资源没有得到充分利用。相反，如果CPU的使用率过高，这可能表明系统的CPU资源已经接近或达到其处理能力的极限，从而导致性能瓶颈。在这种情况下，可能需要考虑升级硬件或优化任务分配，以确保系统的高效运行。因此，合理监控和分析CPU的使用率对于优化系统性能和资源管理至关重要。

是否还记得我们在系统中配置的空闲任务？我们在特定的时间间隔内对这个空闲任务进行计时，通过测量它所占用的时间，然后将这个时间除以总时间，从而可以计算出CPU的空闲率。这个空闲率揭示了在给定的时间段内，CPU未执行任何有效任务而处于闲置状态的时间比例。空闲率与使用率之和等于 1。利用这种方法，我们可以有效地监控和评估CPU的使用状况，进而对系统性能进行优化和调整。

在Cortex-M3 处理器中，我们已经配置并启用了SysTick定时器。那么，是否还存在其他可供利用的定时器呢？据我所了解，实际上并没有其他额外的定时器可供使用。但是不要担心，Cortex-M3 为开发者提供了一个跟踪组件DWT（Data Watchpoint and Trace，即数据监视点和跟踪单元），它是Cortex-M3 的一个调试组件。在DWT中，包含了一个 32 位的时钟周期计数器DWT_CYCCNT，它能够对内核的时钟周期进行精确计数，因此其精度可以达到纳秒级别，非常之高。

通过使用DWT在特定时间段内测量空闲任务的运行时间，我们可以计算出CPU的使用率。为了实现这一目标，我们首先需要确定一个基准时间，这将作为CPU利用率监测周期的基础。我们采用SysTick定时器来实现这一功能，例如，利用SysTick定时器设置一个 100 毫秒的监测周期。在这个周期内，我们通过DWT记录空闲任务的执行时间，从而实现对CPU使用率的监测。

值得注意的是，在我执行的测试中，DWT对时钟周期的计数似乎偏高，我

推测这可能是由于CPU的运行速度经过了优化。因此，我决定对先前的测试方法进行微调。虽然DWT的计时偏高，但是我们的SysTick（System Timer）的计时是准确的。我们在系统启动之前全速运行 100ms，记录下在这个 100ms内DWT的时钟个数，这个 100ms的时间段由SysTick提供。接下来，通过DWT对空闲任务的计时，我们可以得出空闲任务时钟个数。将空闲任务时钟个数除以 100ms内DWT的时钟个数，也可以得到CPU的空闲率，同时这个方法也不会受到系统时钟的影响。

要使用DWT，我们需要设置调试乃及监视器控制寄存器 DEMCR（：0xE000EDF8）的第 24 位，该位为跟踪系统的使能位，如要使用DWT，该位必须设置为 1。

同时，我们必须启用和停止DWT计时功能，让我们来观察一下DWT控制寄存器（地址为 0xE0001000）。

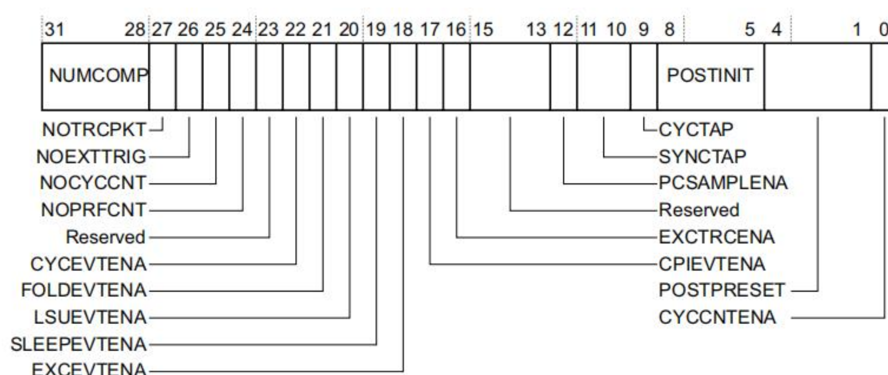

图 51 DWT控制寄存器

在DWT控制寄存器中，我们主要关注的是第 0 位，即CYCCNTENA位。当该位被设置为 0 时，DWT计时功能将被关闭；而当它被设置为 1 时，DWT计时功能则会被激活。计时器的值存储在DWT_CYCCNT（地址为 0xE0001004）寄存器中。

12.2 CPU使用率实现

打开jd_cm3.s，对相关寄存器进行定义。

```
JD_DEMCR                EQU 0xE000EDFC     ;DEMCR的地址 用于使能DWT
JD_DWT_CYCCNT           EQU 0xE0001004     ;DWT计数寄存器
JD_DWT_CTRL             EQU 0xE0001000     ;DWT控制寄存器
```

基于上述分析，我们着手开始编写用于初始化DWT、重置DWT计时器、启动DWT计时、停止DWT计时以及获取DWT计时结果的程序代码。

```
jd_asm_dwt_init          PROC      ;DWT初始化
                         EXPORT jd_asm_dwt_init

                         LDR R0,=JD_DEMCR ;使能DWT
                         LDR R1,[R0]
                         ORR R1,#0x1000000
                         STR R1,[R0]

                         BX LR
                         ENDP


jd_asm_dwt_set0          PROC
                         EXPORT    jd_asm_dwt_set0

                         LDR R0,=JD_DWT_CYCCNT ;DWT清 0
                         MOV R1,#0
                         STR R1,[R0]

                         BX LR
                         ENDP

jd_asm_dwt_start         PROC      ;DWT计时开始
                         EXPORT    jd_asm_dwt_start

                         LDR R0,=JD_DWT_CTRL         ;DWT启动计时
                         LDR R1,[R0]
                         ORR R1,#0x1
                         STR R1,[R0]

                         BX LR
                         ENDP

jd_asm_dwt_stop          PROC      ;         DWT停止计时
                         EXPORT    jd_asm_dwt_stop

                         LDR R0,=JD_DWT_CTRL         ;DWT启动计时
                         LDR R1,[R0]
                         AND R1,#0xFFFFFFFE
                         STR R1,[R0]
```

```
                        BX LR
                        ENDP

jd_asm_dwt_get          PROC      ;DWT计时获取
                        EXPORT    jd_asm_dwt_get

                        LDR R0,=JD_DWT_CYCCNT
                        LDR R0,[R0]

                        BX LR
                        ENDP
```

在所提供的代码段中，函数jd_asm_dwt_init负责初始化DWT，其操作是将DEMCR寄存器的第 24 位设置为 1。接着，jd_asm_dwt_set0 函数用于重置DWT的周期计数器，通过清除DWT_CYCCNT寄存器实现。函数jd_asm_dwt_start则用于启动DWT的计时功能，通过将DWT控制寄存器的第 0 位设置为 1 来实现。相对应的，jd_asm_dwt_stop函数用于停止计时，其操作是将DWT控制寄存器的第 0 位清零。最后，jd_asm_dwt_get函数用于读取DWT_CYCCNT寄存器的值，即获取当前的计时数据，该数据通过寄存器R0 返回。

新建一个文件jd_cpu_u.c，存放我们的CPU使用率监测代码。

```
#include "jdos.h"
// 标志位
#define U_FLAG 1
jd_int32_t jd_cpu_u = 0;
jd_uint8_t jd_cpu_u_flag = 0;
extern void jd_asm_dwt_init(void);
extern void jd_asm_dwt_start(void);
extern void jd_asm_dwt_stop(void);
extern jd_uint32_t jd_asm_dwt_get(void);
extern void jd_asm_dwt_set0(void);
/**
 * @description: cpu利用率初始化  得到最大运行时间
 * @return {*}
 */void jd_cpu_u_init(void){
      jd_asm_dwt_init();
      jd_asm_dwt_set0();
      jd_asm_dwt_start();
}// （1）
/**
 * @description: cpu计时逻辑
 * @return {*}
```

```
 */void jd_cpu_u_start_stop(void){
     if (jd_cpu_u_flag == U_FLAG) // （2）
     {
             if (jd_task_runing == jd_task_frist) // （3）
             {
                     jd_asm_dwt_start();
             }
             Else // （4）
             {
                     jd_asm_dwt_stop();
             }
     }
}
/**
 * @description: 返回CPU利用率
 * @return {*}
 */jd_uint32_t jd_cpu_u_get(void){
     return jd_cpu_u;// （5）
}
/**
 * @description: 外部1ms周期性调用
 * @return {*}
 */void jd_cpu_u_ctr(void){
     static jd_uint8_t jd_cpu_time_100ctr = 0;// （6）
     static jd_uint32_t jd_cpu_100max = 0;// （7）

     if (jd_time == 100&&jd_cpu_u_flag!=U_FLAG)// （8）
     {
             jd_cpu_u_flag = U_FLAG;// （9）
             jd_cpu_100max = jd_asm_dwt_get();// （10）
             jd_cpu_u = jd_cpu_100max;// （11）
     }
     if (jd_cpu_u_flag == U_FLAG)// （12）
     {
             if (++jd_cpu_time_100ctr == 100)// （13）
             {
                     jd_cpu_u = 100 - (float)jd_asm_dwt_get() / jd_cpu_100max *
100;// （14）
                     jd_printf("jd_cpu_u:%d%%\r\n", jd_cpu_u);// （15）

                     jd_cpu_time_100ctr = 0;// （16）
                     jd_asm_dwt_stop();// （17）
                     jd_asm_dwt_set0();// （18）
             }
```

```
        }
    }
```

在上述代码中（1）调用DWT初始化，将DWT_CYCCNT清 0，启动DWT计时；

（2）判断是标志是否开启；

（3）判断切换的任务是否是空闲任务，如果是空闲任务则开启计时；

（4）如果不是空闲任务，则关闭计时；

（5）返回CPU使用率的数据；

（6）创建一个用于循环使用的变量；

（7）创建一个保存 100ms内全速运行的时钟个数的变量；

（8）判断系统时间达到第 100ms且标志没有开启；

（9）如果达到 100ms则开启标志；

（10）如果达到 100ms则获取此时全速运行的时钟个数；

（11）没有实际意义；

（12）确保标志开启；

（13）每 1ms调用一次，直至 100ms；

（14）CPU使用率=100-（空闲任务时钟个数/100ms全速运行时钟个数*100)；

（15）打印CPU使用率；

（16）100ms变量清 0；

（17）停止DWT计时；

（18）DWT计时清 0；

我们已经成功完成了对CPU使用率监控功能的代码编写，并且接下来计划将这些相关函数的调用嵌入到相应的代码位置。

将`jd_cpu_u_init()`函数集成至`jd_init()`函数中以完成初始化流程。同时，将`jd_cpu_u_start_stop()`函数嵌入到所有上下文切换的处理环节，当前系统中这一操作发生在两个特定位置：SVCall和PendSV异常处理代码段。

```
59  /**
60   * @description: PendSV处理函数
61   * @return {*}
62   */
63  void jd_PendSV_Handler(void)
64  {
65    jd_task_t *jd_task;
66
67    jd_asm_cps_disable();
68
69    // 获取数据域
70    jd_task = (jd_task_t *)jd_task_list_readying; // 获取任务数据
71    jd_task_runing->status = JD_TASK_READY;
72    // 任务暂停或延时状态，或者当前任务优先级低，当前任务放弃cpu使用权
73    jd_task->status = JD_TASK_RUNNING;             // 即将运行的任务改为正在运行状态
74    jd_task_stack_sp = &jd_task_runing->stack_sp;      // 更新当前任务全局栈指针变量
75    jd_task_runing = jd_task;                   // 更改当前为运行的任务
76    jd_task_next_stack_sp = &jd_task_runing->stack_sp; // 更新下一个任务全局栈指针变量
77
78    jd_cpu_u_start_stop();
79
80    jd_asm_cps_enable();
81
82    jd_asm_pendsv_handler(); // 切换上下文
83  }
84
85  /**
86   * @description: SVC处理函数
87   * @return {*}
88   */
89  void jd_SVC_Handler(void)
90  {
91    jd_cpu_u_start_stop();
92
93    jd_asm_svc_handler();
94  }
95
```

图 52 jd_cpu_u_start_stop加入位置

这里的jd_SVC_Handler只是将SVCall中原来的函数抽离出来，方便管理。在stm32f1xx_it.c文件中的代码如下所示，该操作旨在优化代码结构，提高代码的可读性和可维护性。

```
void SVC_Handler(void)
{
  /* USER CODE BEGIN SVCall_IRQn 0 */
  extern void jd_SVC_Handler(void);
  jd_SVC_Handler();

  /* USER CODE END SVCall_IRQn 0 */
  /* USER CODE BEGIN SVCall_IRQn 1 */

  /* USER CODE END SVCall_IRQn 1 */
}
```

图 53 jd_SVC_Handler与SVC异常绑定

最终步骤，我们将jd_cpu_u_ctr集成至SysTick定时器中，以周期性方式调用，从而计算出CPU的空闲周期。

```c
void HAL_IncTick(void)
{
  jd_asm_cps_disable();
  uwTick += uwTickFreq; // 系统自带不可删除,否则hal_delay等hal库函数不可用

  jd_time++; // jd_lck++
  // 判断延时表头是否到达时间, 若没有到达时间, 则切换, 若到达时间则将任务加入到就绪任务,再切换任务
  jd_task_t *jd_task;
  jd_task = (jd_task_t *)jd_task_list_delaying; // 获取任务数据

  while (jd_task->timeout == jd_time)
  {
    // 如果循环定时器任务, 将下一次定时时间写入任务信息
    if (jd_task->timer_loop == JD_TIMER_LOOP)
    {
      jd_task->timeout = jd_time + jd_task->timer_loop_timeout;
    }

    jd_task_list_delaying = jd_node_delete(jd_task_list_delaying, jd_task_list_delaying); // 删除延时完成的节点

    jd_task->status = JD_TASK_READY; // 将任务更改为就绪状态
    // 加入就绪链表
    jd_task_list_readying = jd_node_in_rd(jd_task_list_readying, &jd_task->node);

    if (jd_task_list_delaying == JD_NULL)
      break;
    jd_task = (jd_task_t *)jd_task_list_delaying; // 获取任务数据
  }


  // 这里计算的是空闲任务的运行时间
  jd_cpu_u_ctr();


  jd_asm_cps_enable();

  jd_asm_pendsv_putup();
}
```

图 54 jd_cpu_u_ctr加入位置

将代码编译后下载至开发板，通过串口连接，我们可以观察到当前CPU的使用率仅为 5%，这表明我们的系统仅占用了CPU资源的 5%。

```
jd_cpu_u:5%
jd_cpu_u:5%
jd_cpu_u:5%
jd_cpu_u:5%
jd_cpu_u:5%
jd_cpu_u:5%
jd_cpu_u:5%
jd_cpu_u:5%
```

图 55 CPU使用率 5%

让我们进行测试，在测试任务中引入以下代码。

```c
25  void task2()
26  {
27
28    while (1)
29    {
30      // jd_printf("task2\r\n");
31      HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_7);
32      HAL_Delay(20);
33      jd_delay(80);
34
35    };
36  }
```

图 56 CPU使用率测试

在代码测试中，我们采用HAL_Delay函数来实现对程序阻塞运行 20 毫秒。与此同时，我们还使用了jd_delay函数来实现系统的延时，延时的时间为 80 毫秒。根据理论计算，当这两个延时函数同时作用时，CPU的使用率应是 20%。为了验证这一点，我们对代码进行了重新编译，并将其下载到开发板中进行实际测试。通过观察开发板上的运行情况，我们发现CPU的使用率达到了 25%。这一结果充分证明了我们所实现的CPU使用率监测功能是成功的。

```
jd_cpu_u:25%
jd_cpu_u:25%
jd_cpu_u:25%
jd_cpu_u:25%
jd_cpu_u:25%
jd_cpu_u:25%
```

图 57 CPU 使用率 25%

第十三章 优化内存管理

13.1 优化原理

在我们之前编写的代码中，我们采用了数组这种数据结构来实现内存管理的功能。通过使用数组，我们可以相对简单地分配出一段连续的内存空间，以便于存储和管理数据。然而，数组这种数据结构有一个显著的缺点，那就是它所占用的内存空间是固定的。在我们的STM32F103RCT6 微控制器中，拥有高达 48K字节的RAM空间，这是一个相当可观的资源。然而，在之前的内存管理方案中，我们并没有充分利用这些宝贵的内存资源，导致大量可用的内存空间被闲置，没有得到有效的利用。

我们需要充分利用这 48K字节的RAM空间，这样一来，我们能够显著增加可用的内存资源，从而提高程序的灵活性和效率。通过优化内存管理，我们可以更好地利用STM32F103RCT6 微控制器的硬件资源，使其在处理复杂任务时更加得心应手。

在着手优化代码之前，我们首先需要彻底理解代码和数据在Cortex-M3 处理器的内存空间中是如何分布和组织的。这包括了解不同类型的内存区域，例如程序存储区、堆区、栈区以及全局变量区等，以及它们各自的作用和特点。此外，我们还需要掌握内存映射的具体细节，比如各个内存段的起始地址和大小，以及它们之间的相互关系。通过对这些内存分布情况的深入了解，我们可以更好地识别出潜在的瓶颈和优化点，从而有针对性地进行代码优化，提高程序的性能和效率。

众所周知，Cortex-M3 处理器的地址空间达到了 4GB的容量。这一特性是由于Cortex-M3 采用了 32 位的架构设计。32 位处理器意味着它可以处理和寻址的数据宽度为 32 位，从而使得其能够访问高达 2 的 32 次方个不同的地址。因此，Cortex-M3 能够支持一个庞大的地址空间，即 4GB，这为嵌入式系统提供了丰富的资源和灵活性，使其能够应对各种复杂的应用需求。

让我们看一下这些空间的具体定义。

图 58 4GB空间定义

可以看到这些空间被简单的定义成了几个区域，分别是Cortex-M3 的外设区(0xE0000000-0xFFFFFFFF)、片外外设区（0xA000000-0xDFFFFFFF）、片外RAM区（0x60000000-0x9FFFFFFF）、片上外设区（0x40000000-0x5FFFFFFF）、片上SRAM（0x20000000-0x3FFFFFFF）和代码区（0x00000000-0x1FFFFFFF）。

外设区：存放外设寄存器的区域，这些寄存器控制着Cortex-M3 处理器的外设功能，通过这些寄存器，开发者可以配置和管理各种外设，如定时器、串口、ADC等。

片外外设区：是处理器与外部设备进行交互的重要区域，它允许处理器访问和控制连接到微控制器上的各种外设，这个区域通常用于映射那些不直接集成在芯片内部的外设，比如外部存储器、通信接口和其他专用硬件模块。

片外RAM区：处理器与外部设备进行交互的重要区域，它允许处理器访问和控制连接到微控制器上的各种外设，这个区域通常用于映射那些不直接集成在芯片内部的外设，比如外部存储器、通信接口和其他专用硬件模块。

片上外设区：提供了对片上外设的直接访问，使得微控制器能够更加高效地处理数据和控制信号，这种集成的外设区域通常包括定时器、串行通信接口、模拟数字转换器（ADC）等。

片上SRAM区：提供了快速的数据存储和访问，是微控制器内部用于临时存储数据的重要区域。

代码区：设计为存放程序代码的区域。

在掌握了Cortex-M3 的基本内存分区之后，我们将深入探讨代码区域以及片上SRAM区域。

我们编写的程序下载到了哪个区呢？它是如何运行起来的呢？实际上一般程序都会被下载到指定的代码区，而数据将会加载到片上SRAM区。

具体位置会因不同的半导体制造商而异。以STM32 为例，参考其官方手册，我所使用的STM32F103RCT6 芯片的内部SRAM区域起始地址为 0x20000000，而Flash的起始地址为 0x08000000，那我们之前编译的代码下载后应该存放在0x08000000 地址之后的区域，而单片机运行起来后的数据则存储在 0x20000000后的区域。

Flash中包含文本段（代码和常量）和只读数据段等，我们这里不深入讨论。

内部SRAM区中包含data段、bss段、Heap（堆）段、Stack（栈）段，这些段过后就是剩余空间。

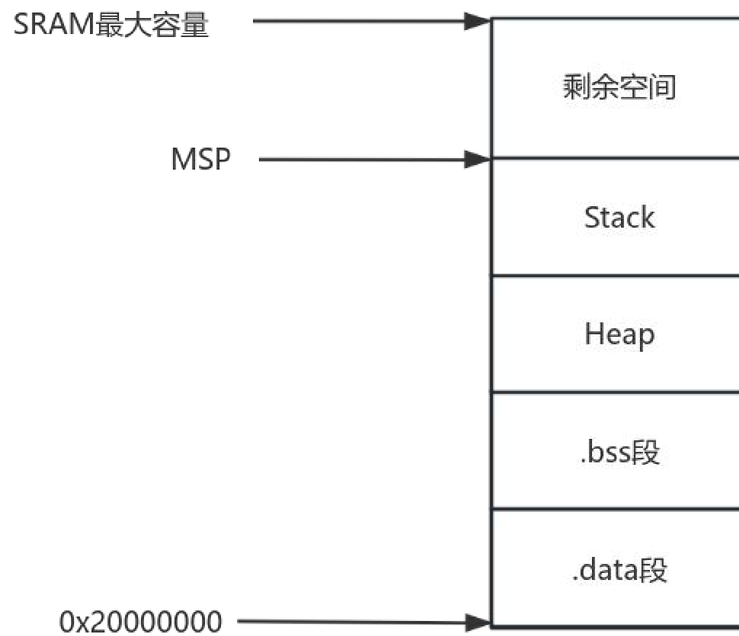

图 59 SRAM区空间分布

data段用于存储那些初始化时非零的全局变量；而未初始化的全局变量或初始化为零的全局变量则存放在bss段。Heap和Stack段，我们可以通过查看启动文件来了解它们的具体情况。

关于启动文件在之前的章节中有所介绍，现在让我们从内存的监督从新审视一下启动文件。

```
32  Stack_Size      EQU     0x400
33
34                  AREA    STACK, NOINIT, READWRITE, ALIGN=3
35  Stack_Mem       SPACE   Stack_Size
36  __initial_sp
37
38  ; <h> Heap Configuration
39  ;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
40  ; </h>
41
42  Heap_Size       EQU     0x200
43
44                  AREA    HEAP, NOINIT, READWRITE, ALIGN=3
45  __heap_base
46  Heap_Mem        SPACE   Heap_Size
47  __heap_limit
48
49                  PRESERVE8
50                  THUMB
51
52
53  ; Vector Table Mapped to Address 0 at Reset
54                  AREA    RESET, DATA, READONLY
55                  EXPORT  __Vectors
56                  EXPORT  __Vectors_End
57                  EXPORT  __Vectors_Size
58
59  __Vectors       DCD     __initial_sp               ; Top of Stack
60                  DCD     Reset_Handler              ; Reset Handler
61                  DCD     NMI_Handler                ; NMI Handler
62                  DCD     HardFault_Handler          ; Hard Fault Handler
63                  DCD     MemManage_Handler          ; MPU Fault Handler
64                  DCD     BusFault_Handler           ; Bus Fault Handler
65                  DCD     UsageFault_Handler         ; Usage Fault Handler
```

图 60 启动文件

Stack_Size EQU 0x400：定义了一个宏Stack_Size，其值为 0x400（1024 字节），这表示堆栈的大小；

AREA STACK, NOINIT, READWRITE, ALIGN=3：定义了一个名为STACK的内存区域，它不会被初始化，可读写，并且按照 3 的对数（即 8 字节）对齐；

Stack_Mem SPACE Stack_Size：在STACK区域中分配了Stack_Size指定大小的空间作为堆栈；

__initial_sp：这是一个标签，用于标记堆栈的初始栈指针位置；

Heap_Size EQU 0x200：定义了一个宏Heap_Size，其值为 0x200（512 字节），这表示堆的大小；

AREA HEAP, NOINIT, READWRITE, ALIGN=3：定义了一个名为HEAP的内存区域，它不会被初始化，可读写，并且按照 3 的对数（即 8 字节）对齐；

__heap_base：这是一个标签，用于标记堆的起始地址；

Heap_Mem SPACE Heap_Size：在HEAP区域中分配了Heap_Size指定大小的空间作为

堆。

__heap_limit：这是一个标签，用于标记堆的结束地址；

PRESERVE8：这是一个伪指令，用于告诉链接器保留接下来的 8 个字节；

THUMB：这指定了代码将以THUMB模式运行，这是一种 16 位的指令集，用于减少代码的大小；

后面的代码：定义了向量表。

我们观察到上述代码中定义的段落，在第 34 行，STACK关键字标识了一个STAC段；在第 44 行，HEAP关键字标识了一个HEAP段；而在第 54 行，RESET关键字标识了一个重置（RESET）段。

我们无需深入了解这些段通过什么方式最后在内存中这样分布，因为这一过程将由编译器自动完成。我们的关注点应放在STACK段之后的剩余空间上，我们需要充分利用这些空间，为了确定剩余空间的起始地址，我们必须知道STACK段的地址。

现在，让我们回到启动文件的第 35 行，这里定义了一个名为Stack_Mem的变量，它代表了STACK段的起始地址。而STACK段的大小Stack_Size已在第 32 行被定义为 0x400 字节。因此，STACK段的结束地址可以通过Stack_Mem加上Stack_Size来计算得出。

为了实现内存管理，我们不仅需要知道剩余空间的起始地址，还必须了解其SRAM具体大小。不同芯片的SRAM容量各异。以我手中的STM32F103RCT6 芯片为例，其内部SRAM容量为 48K字节，SRAM的结束地址为 0xC000。因此，剩余空间的大小可以通过计算得出：SRAM的结束地址-（STACK段的结束地址-SRAM的起始地址）。如果你对所使用芯片的SRAM容量不够熟悉，可以在Keil设置中查看相关信息。

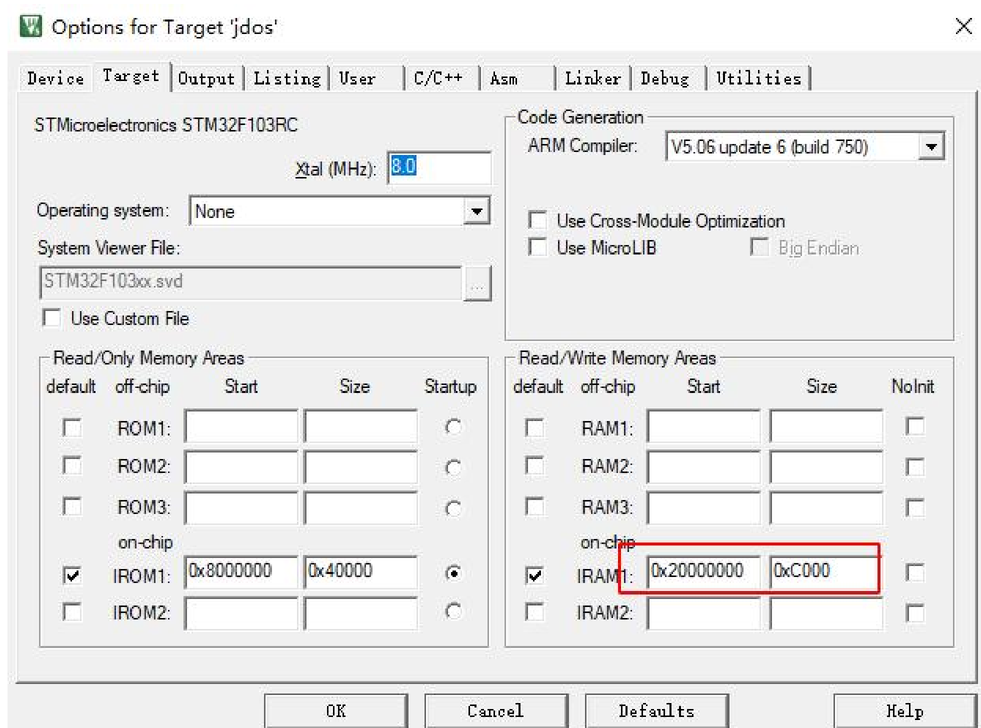

图 61 Keil显示SRAM容量

13.2 优化内存管理实现

既然已经理解了原理，接下来就让我们着手编写代码吧，首先将启动文件中的Stack_Siz和Stack_Mem共享出去。

```
32  Stack_Size      EQU     0x400
33      EXPORT Stack_Size
34
35                  AREA    STACK, NOINIT, READWRITE, ALIGN=3
36  Stack_Mem       SPACE   Stack_Size
37      EXPORT Stack_Mem
38  __initial_sp
```

图 62 启动文件共享信息

接着我们在jd_cm3.s中添加以下代码，用于计算Stack段的结束地址。

```
jd_initial_sp_get         PROC
                          EXPORT jd_initial_sp_get

                          LDR R0,=Stack_Mem
                          LDR R1,=Stack_Size
                          ADD R0,R1

                          BX LR
                          ENDP
```

最后我们在jd_memory.c中修改内存的初始化。

```
#define JD_MEM_SIZE 0xC000  //（1）
#define JD_CPU_START_MEM 0x20000000  //（2）
extern jd_uint32_t jd_initial_sp_get(void);
jd_mem_t *jd_mem_use = JD_NULL;
jd_uint32_t jd_mem_space; //（3）
/**
 * @description: 内存初始化
 * @return {*}
 */
jd_uint32_t jd_mem_init(){
	jd_mem_space = jd_initial_sp_get(); //（4）
	jd_mem_use = (jd_mem_t *)jd_mem_space; // 传入内存块地址
	// jd_mem_use->node.addr = jd_mem_use;    // 保存内存块地址

	jd_mem_use->node.next = JD_NULL;
	jd_mem_use->node.previous = JD_NULL;

	jd_mem_use->used = JD_MEM_FREE;      // 初始为空闲内存
	jd_mem_use->mem_size = JD_MEM_SIZE-(jd_mem_space-JD_CPU_START_MEM); // 初始
内存块大小  //（5）

	return JD_OK;
}
```

在上述代码中（1）定义SRAM大小；

（2）定义SRAM起始地址；

（3）将原来的数组定义修改为普通的参数用于存储剩余空间起始地址；

（4）获取剩余空间起始地址；

（5）计算剩余空间的大小。

我们顺便添加一个函数，用于获取已使用的内存总量。该函数遍历所有任务的内存占用情况，并将它们相加以计算出总的已使用内存大小。

```
/**
 * @description: 获取已经使用的空间大小
 * @return {*}
 */
jd_uint32_t jd_mem_used_get()
{
	jd_mem_t *jd_mem_temp;
	jd_uint32_t jd_mem_used;
	jd_mem_temp = jd_mem_use;
```

```
        jd_mem_used = jd_mem_space-JD_CPU_START_MEM;
        //遍历所有内存空间
        while(1)
        {
                if(jd_mem_temp->used==JD_MEM_USED)
                jd_mem_used += jd_mem_temp->mem_size;
                if(jd_mem_temp->node.next==JD_NULL)
                        break;
                else
                        jd_mem_temp = (jd_mem_t *)jd_mem_temp->node.next;
                }
        jd_printf("used_mem/all_mem:%dKB/%dKB\r\n",jd_mem_used/1000,JD_MEM_SIZE/10
00);
        return jd_mem_used;
}
```

我们在测试任务中添加测试代码。

```
void task3(){
        while (1)
        {
                jd_mem_used_get();
                jd_uint32_t *test_sp1 = jd_malloc(1024*20);
                jd_mem_used_get();
                jd_uint32_t *test_sp2 = jd_malloc(1024*10);
                jd_mem_used_get();
                jd_delay(320);
                jd_free(test_sp1);
                jd_mem_used_get();
                jd_free(test_sp2);
                jd_mem_used_get();
                HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_7);
                jd_mem_used_get();
        };
}
```

以上代码分别申请 20k内存和 10k的内存，在 320ms后分别进行释放，在这期间我们检测内存容量的变化。

```
used_mem/all_mem:3KB/49KB
used_mem/all_mem:24KB/49KB
used_mem/all_mem:34KB/49KB
used_mem/all_mem:14KB/49KB
used_mem/all_mem:3KB/49KB
```

图 63 内存测试

初始占用 3KB的内存空间，当申请 20KB的额外内存时，代码能够识别出总共 24KB的内存已被使用。随后，若再申请 10KB内存，系统将检测到 34KB的内存已被占用。在释放这些内存后，已使用的内存总量将回落至初始状态。

在测试显示的 3K内存中，一部分被分配给了数据段（data）、未初始化数据段（bss）、堆（Heap）和栈（Stack），这些区域共占用 1K字节。剩余部分则用于任务管理，包括一个空闲任务和三个测试任务，每个任务分配了 512 字节的空间，因此空闲任务和三个测试任务总共占用 2K字节，在不额外申请内存的情况下，目前的内存使用量总计为 3K字节。

上述证实了我们对内存管理的优化已取得成效，成功将芯片上的全部可用的SRAM空间纳入管理范畴，并且验证了先前的内存管理设计的合理性

最后

在本篇文章中，我们编写了一个简单实时操作系统jdos，虽然我们已经完成了一些初步设计工作，但在代码优化方面，我们尚未进行深入研究和必要的改进，比如我们的内存管理和任务管理主要依赖于基础的遍历方法，这在实际应用中可能会导致效率问题。

虽然jdos的核心功能已经初具RTOS的雏形，但一些RTOS的关键特性，如任务间的同步与通信等，目前尚未实现，这些功能对于RTOS的稳定性和可靠性至关重要，因此在未来的工作中需要重点关注和补充。

通过阅读本文，我们相信你已经掌握了编写RTOS的基本技能和方法，在此，我衷心感谢各位读者的耐心阅读和持续关注，希望本文能够为您提供有价值的参考和指导，帮助您在RTOS开发的道路上更进一步。

2024.11.07 江小鉴