第1章　前言

1.1 如何学习本书

本书与市面上的任何一本 uC/OS-III 相关的书都不一样，它们要么是翻译官方的参考手册，要么是讲如何使用 uC/OS-III，要么是讲 uC/OS-III 的源码，而本书是教你怎么从 0 开始把 uC/OS-III 写出来，既讲了源码实现，也讲了 API 如何使用。当你拿到本书开始学习的时候你一定会惊讶，原来 RTOS 的学习并没有那么复杂，反而是那么的有趣，原来自己也可以写 OS，成就感立马爆棚。

全书内容循序渐进，不断迭代，前一章都是后一章的基础，必须从头开始阅读，不能进行跳跃式的阅读。在学习的时候务必做到两点：一是不能一味地看书，要把代码和书本结合起来学习，一边看书，一边调试代码。看书倒是很简单，那如何调试代码？即单步执行每一条程序，看看程序的执行流程和执行的效果与自己大脑所想是不是一样；二是在每学完一章之后，必须将配套的例程重写一遍（切记不要复制，哪怕是一个分号，但可以抄），做到举一反三，确保真正理解。在自己写的时候肯定会错漏百出，这个时候要珍惜这些错误，好好调试，这是你提高编程能力的最好的机会。记住，程序不是写出来的，而是调试出来的。

1.2 本书的参考资料

1、 uC/OS-III 官方源代码

2、 uCOS-III 中文翻译 （电子版）

3、 嵌入式操作系统 uCOS-II(第二版) （电子版）

4、 嵌入式实时操作系统 μCOS-II 原理及应用 任哲编著 （电子版）

5、 CM3 权威指南 CnR2 （电子版）

6、 STM32F10xxx Cortex-M3 programming manual （电子版）

1.3 本书的编写风格

本书以 uC/OS-III 官方源码为蓝本，抽丝剥茧，不断迭代，教你怎么从 0 开始把 uC/OS-III 写出来。书中涉及到的数据类型，变量名称、函数名称，文件名称，文件存放的位置都完全按照 uC/OS-III 官方的方式来实现，当你学完这本书之后可以无缝地切换到原版的 uC/OS-III 的使用。要注意的是，在实现的过程中，某些函数我会去掉一些形参和一些冗余的代码，只保留核心的功能，但这并不会影响我们学习。

1.4 本书的配套硬件

本书支持野火 STM32 开发板全套系列，具体型号见表格 1-1，具体图片见图 1-1、图 1-2、图 1-3、图 1-4 和图 1-5。学习的时候如果配套这些硬件平台做实验，学习必会达到事半功倍的效果，可以省去中间硬件不一样时移植遇到的各种问题。

表格 1-1 野火 STM32 开发板型号汇总

型号	区别			
-	内核	引脚	RAM	ROM
MINI	Cortex-M3	64	48KB	256KB
指南者	Cortex-M3	100	64KB	512KB
霸道	Cortex-M3	144	64KB	512KB
霸天虎	Cortex-M4	144	192KB	1MB
挑战者	Cortex-M4	176	256KB	1MB

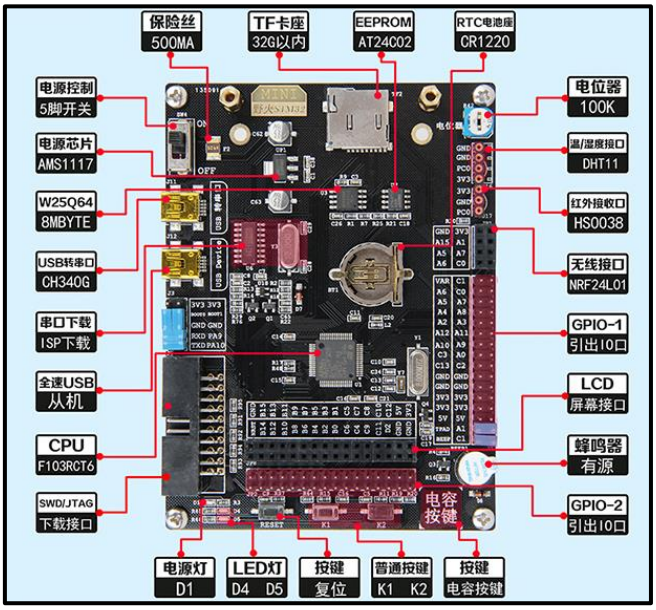


图 1-1 野火【MINI】STM32F103RCT6 开发板

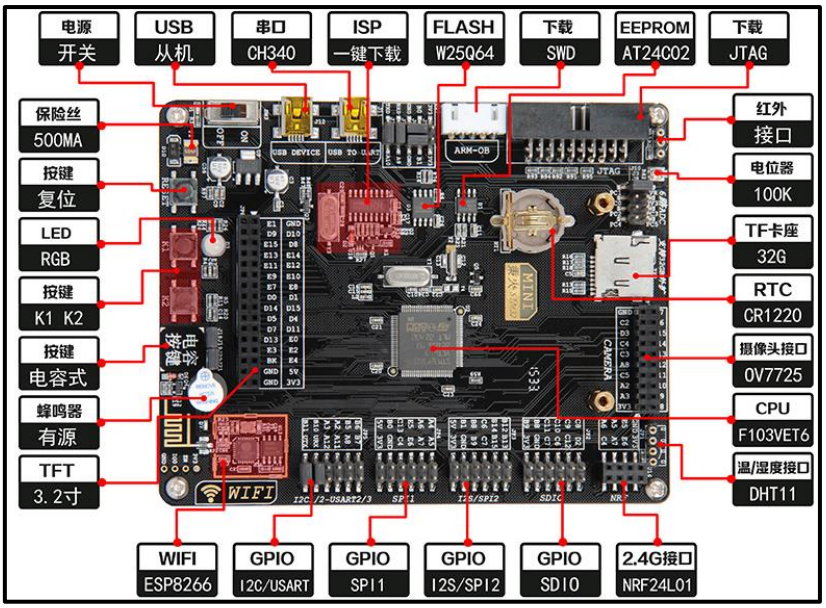


图 1-2 野火【指南者】STM32F103VET6 开发板

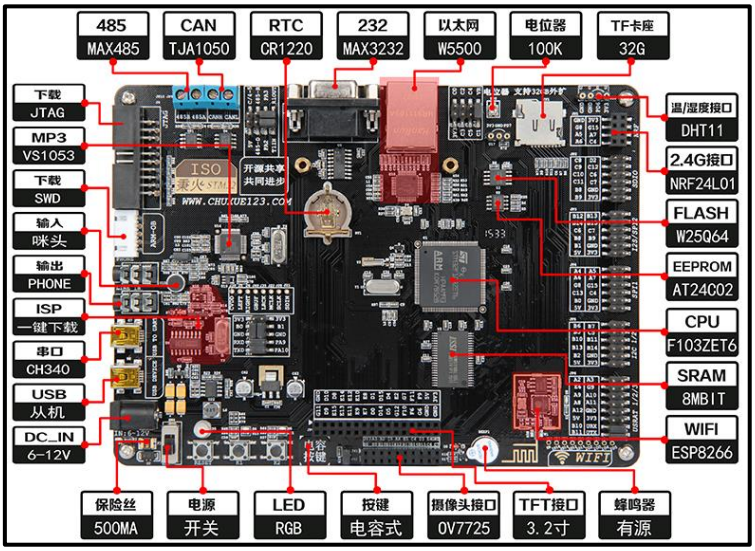


图 1-3 野火【霸道】STM32F103ZET6 开发板

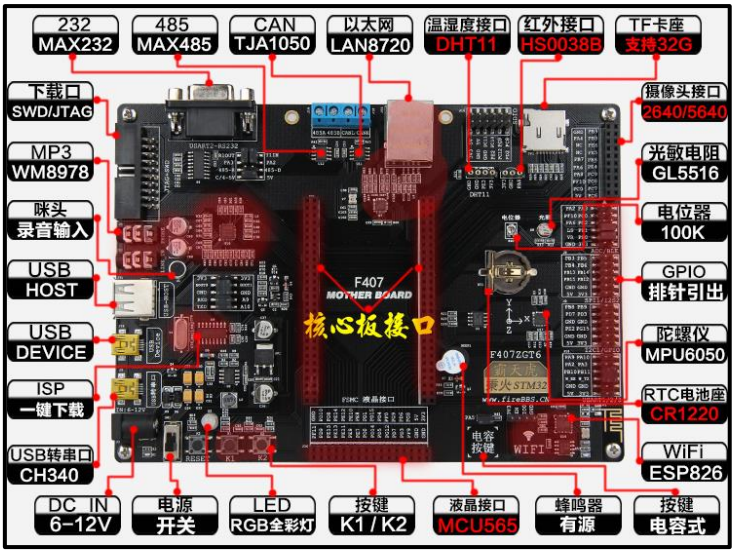


图 1-4 野火【霸天虎】STM32F407ZGT6 开发板

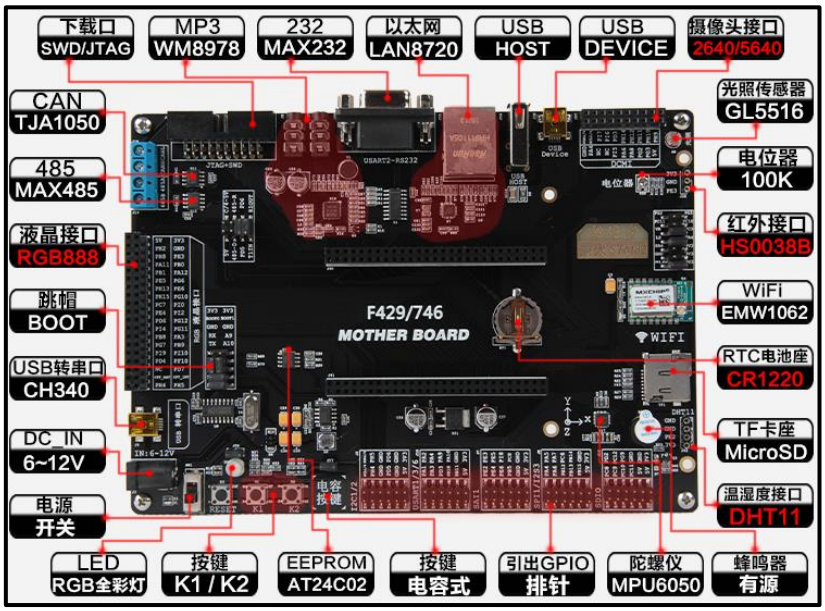


图 1-5 野火【挑战者】STM32F429IGT6 开发板

1.5 本书的技术论坛

如果在学习过程中遇到问题，可以到野火电子论坛：www.firebbs.cn 发帖交流 ，开源共享，共同进步。

鉴于水平有限，本书难免有纰漏，热心的读者也可把勘误发到论坛好让我们改进做得更好，祝您学习愉快，uC/OS-III 的世界，野火与您同行。

第2章 如何学习 RTOS

2.1 为什么要学习 RTOS

当我们进入嵌入式这个领域的时候，往往首先接触的都是单片机编程，单片机编程又首选 51 单片机来入门。这里面说的单片机编程通常都是指裸机编程，即不加入任何 RTOS（Real Time Operation System 实时操作系统）的程序。常用的 RTOS 有国外的 FreeRTOS、ucOS、RTX 和国内的 Huawei LiteOS、和 RT-Thread 等，其中尤以开源且免费的 FreeRTOS 的市场占有率最高，历史悠久的 ucos 屈居第二。

在裸机系统中，所有的程序基本都是自己写的，所有的操作都是在一个无限的大循环里面实现。现实生活中的很多中小型的电子产品用的都是裸机系统，而且也能够满足需求。但是为什么还要学习 RTOS 编程，偏偏还要整个操作系统进来。一是项目需要，随着产品要实现的功能越来越多，单纯的裸机系统已经不能够完美地解决问题，反而会使编程变得更加复杂，如果想降低编程的难度，我们可以考虑引入 RTOS 实现多任务管理，这是使用 RTOS 的最大优势。二是学习的需要，必须学习更高级的东西，实现更好的职业规划，为将来走向人生巅峰迎娶白富美做准备，而不是一味的在裸机编程上面死磕。作为一个合格的嵌入式软件工程师，学习是永远不能停止的事，时刻都在为将来准备。书到用时方恨少，我希望机会来临时你不要有这种感觉。

为了帮大家理清 RTOS 编程的套路，下面我们简单的分析下这两种编程方式的区别，这个区别我称它为学习 RTOS 的命门，只要打通这个任督二脉，以后的 RTOS 学习可以说是易如反掌。在讲解这两种编程方法的区别的时候，我们主要讲方法论，不会涉及到具体的代码编程，主要还是通过伪代码来讲解。

2.2 如何学习 RTOS

裸机编程和 RTOS 编程的风格有些不一样，而且有很多人说 RTOS 的学习很难，这就导致学习的人一听到 RTOS 编程就在心理面忌惮三分，结果就是出师未捷身先死。

那么到底如何学习一个 RTOS？最简单的就是在别人移植好的系统之上，看看 RTOS 里面的 API 使用说明，然后调用这些 API 实现自己想要的功能即可。完全，不用关心底层的移植，这是最简单快速的入门方法。这种方法各有利弊，如果是做产品，好处是可以快速的实现功能，将产品推向市场，赢得先机，弊端是当程序出现问题的时候，因对这个 RTOS 不够了解，会导致调试困难，焦头烂额，一筹莫展。如果是学习，那么只会简单的调用 API，那是不可取的，我们应该深入的学习其中一款 RTOS。

目前市场上现有的 RTOS，它们的内核实现方式都差不多，我们只需要深入学习其中一款就行。万变不离其宗，以后换到其它型号的 RTOS，使用起来，那自然是得心应手。那如何深入的学习一款 RTOS？这里有一个最有效也是最难的方法，就是阅读 RTOS 的源码，深究内核和每个组建的实现方式，这个过程枯燥且痛苦。但为了能够学到 RTOS 的精华，你不入地狱谁入地狱？

市面上虽然有一些讲解相关 RTOS 源码的书，如果你基础不够，且先前没有使用过该款 RTOS，那么源码看起来还是会非常枯燥，且并不能从全局掌握整个 RTOS 的构成和实现。

现在，我们采用一种全新的方法来教大家学习一款 RTOS，即不是单纯的讲里面的 API 如何使用，也不是单纯的拿里面的源码一句句来讲解。而是，从 0 开始，层层叠加，不断地完善，教大家怎么把一个 RTOS 写 0 到 1 写出来，让你在每一个阶段都能享受到成功的喜悦。在这个 RTOS 实现的过程中，只需要你具备 C 语言的基础就行，然后就是跟着我们这个教程笃定前行，最后定有所成。

这个用来教学的 RTOS，我们不会完全自己写一个，不会再重复的造轮子，而是选取年龄最大（26 岁）、商业化最成功、安全验证最多的 uC/OS-III 为蓝本，将其抽丝剥茧，层层叠加，从 0 到 1 写出来。在实现的过程中，数据类型、变量名、函数名称、文件类型等都完全按照 uC/OS-III 里面的写法，不会自己再重新命名。这样学完我们这个课程之后，就可以无缝地过度到 uC/OS-III 的使用。

2.3 如何选择 RTOS

选择一个 RTOS 要看下你是学习还是做产品，如果是学习则选择一个年龄最大，商业化最成功，安全验证最多的来学习，而且是深入学习。那么符合前面这几个标准的只有 ucos，所以，学一个 RTOS，首选 ucos，而且 ucos 的资料是最多的。当然，选择其他的 RTOS 来学习也是可以的。学完之后就是要用了，如果是产品中使用 ucos 就要面临授权的问题，就是要给版权费。一听到要给钱，大家肯定是不乐意了，所以开源免费的 FreeRTOS 就受到各个半导体厂商和开发者的青睐。目前，FreeRTOS 是市场占有率最高的 RTOS，非常适合用来做产品。另外，国内的 RT-Thread 也在迅速的崛起，同样是开源免费。

第3章 新建工程—软件仿真

在开始写 RTOS 之前，我们先新建一个工程，Device 选择 Cortex-M3 内核的处理器，调试方式选择软件仿真，到最后写完整个 RTOS 之后，我们再把 RTOS 移植到野火 STM32 开发板上，到了最后的移植其实已经非常简单，只需要换一下启动文件和添加 bsp 驱动就行。

3.1 新建本地工程文件夹

在开始新建工程之前，我们先在本地电脑端新建一个文件夹用于存放工程。文件夹名字我们取为 RTOS，然后再在该文件夹下面新建各个文件夹和文件，有关这些文件夹的包含关系和作用具体见表格 3-1。

表格 3-1 工程文件夹根目录下的文件夹的作用

文件夹名称			文件夹作用
Doc	-	-	用于存放对整个工程的说明文件，如 readme.txt。通常情况下，我们都要对整个工程实现的功能，如何编译，如何使用等做一个简要的说明。
Project	-	-	用于存放新建的工程文件。
User	uCOS-III	Source	用于存放 uC/OS-III 源码，这里面的代码是纯软件的，跟硬件无关。
		Ports	用于存放接口文件，即 uC/OS-III 与 CPU 连接的文件，也就是我们通常说的移植文件。要想 uC/OS-III 在单片机上面跑起来，这些移植文件必不可少。
	uC-CPU	-	用于存放 uC/OS-III 根据 CPU 总结的通用代码，只跟 CPU 相关。
	uC-LIB	-	用于存放一些 C 语言函数库。
	-	-	用于存放用户程序，如 app.c，main 函数就放在 app.c 这个文件里面。

型号	区别			
-	内核	引脚	RAM	ROM
MINI	Cortex-M3	64	48KB	256KB
指南者	Cortex-M3	100	64KB	512KB
霸道	Cortex-M3	144	64KB	512KB
霸天虎	Cortex-M4	144	192KB	1MB
挑战者	Cortex-M4	176	256KB	1MB

3.2 使用 KEIL 新建工程

开发环境我们使用 KEIL5，版本为 5.15，高于或者低于 5.15 都行，只要是版本 5 就行。

3.2.1 New Progect

首先打开 KEIL5 软件，新建一个工程，工程文件放在目录 Project\RVMDK（uv5）下面，名称命名为 YH-uCOS-III，其中 YH 是野火拼音首字母的缩写，当然你也可以换成其它名称，但是必须是英文，不能是中文，切记。

3.2.2 Select Device For Target

当命名好工程名称，点击确定之后会弹出 Select Device for Target 的选项框，让我们选择处理器，这里我们选择 ARMCM3，具体见图 3-1。

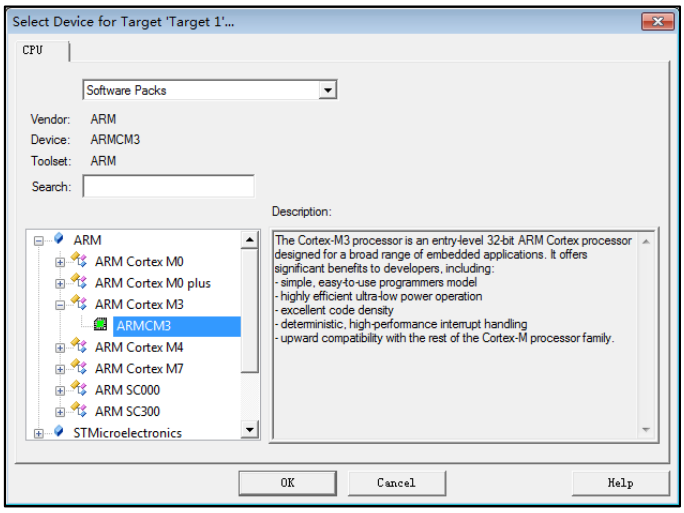


图 3-1 Select Device For Target

3.2.3 Manage Run-Time Environment

选择好处理器，点击 OK 按钮后会弹出 Manage Run-Time Environment 选项框。这里我们在 CMSIS 栏选中 CORE 和 Device 栏选中 Startup 这两个文件即可，具体见图 3-2。

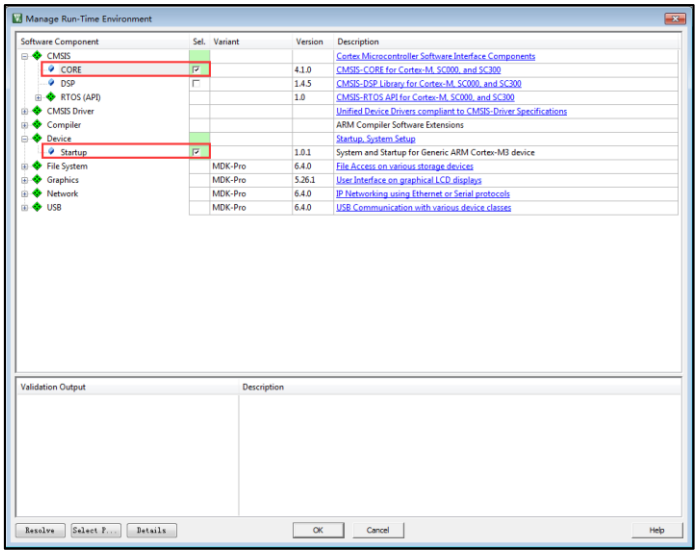


图 3-2Manage Run-Time Environment

点击 OK，关闭 Manage Run-Time Environment 选项框之后，刚刚我们选择的 CORE 和 Startup 这两个文件就会添加到我们的工程组里面，具体见图 3-3。

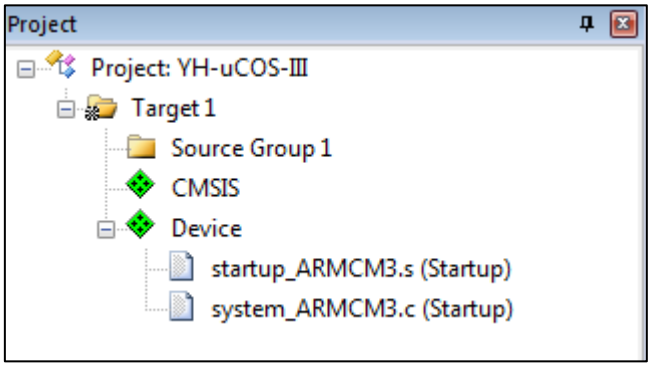


图 3-3CORE 和 Startup 文件

其实这两个文件刚开始都是存放在 KEIL 的安装目录下，当我们配置 Manage Run-Time Environment 选项框之后，软件就会把选中好的文件从 KEIL 的安装目录拷贝到我们的工程目录：Project\RTE\Device\ARMCM3 下面。其中 startup_ARMCM3.s 是汇编编写的启动文件，system_ARMCM3.c 是 C 语言编写的跟时钟相关的文件。更加具体的可直接阅读这两个文件的源码。只要是 Cortex-M3 内核的单片机，这两个文件都适用。

3.3 在 KEIL 工程里面新建文件组

在工程里面添加 User、uC/OS-III Source、uC/OS-III Ports、uC/CPU、uC/LIB 和 Doc 这几个文件组，用于管理文件，具体见图 3-4。

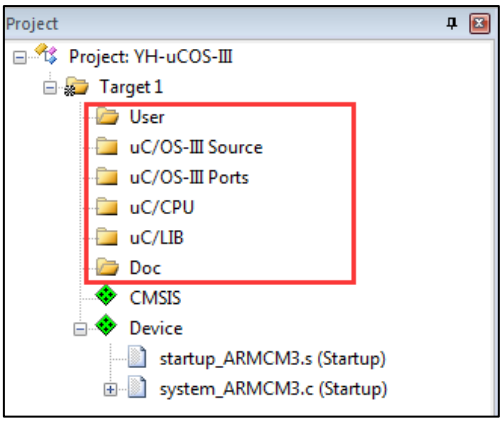


图 3-4 新添加的文件组

对于新手，这里有个问题就是如何添加文件组？具体的方法为鼠标右键 Target1，在弹出的选项里面选择 Add Group…即可，具体见图 3-5，需要多少个组就鼠标右击多少次 Target1。

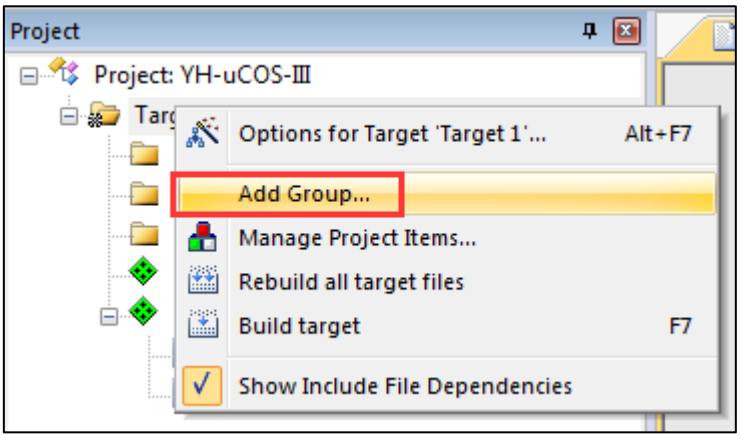


图 3-5 如何添加组

3.4 在 KEIL 工程里面添加文件

在工程里面添加好组之后，我们需要把本地工程里面新建好的文件添加到工程里面。具体为把 readme.txt 文件添加到 Doc 组，app.c 添加到 User 组，至于 OS 相关的文件我们还没有编写，那么 OS 相关的组就暂时为空，具体见图 3-6。

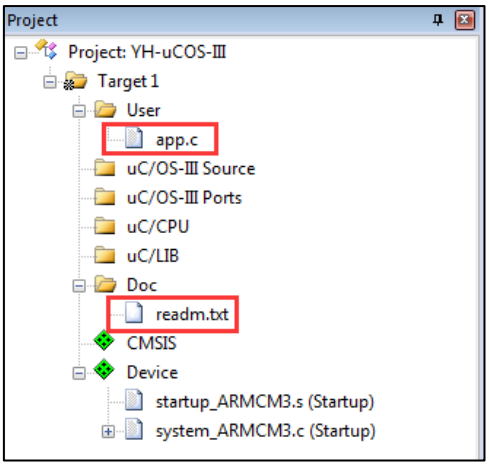


图 3-6 往组里面添加好的文件

对于新手，这里有个问题就是如何将本地工程里面的文件添加到工程组里里面？具体的方法为鼠标左键双击相应的组，在弹出的文件选择框中找到要添加的文件，默认的文件类型是 C 文件，如果要添加的是文本或者汇编文件，那么此时将看不到，这个时候就需要把文件类型选择为 All Files，最后点击 Add 按钮即可，具体见图 3-7。

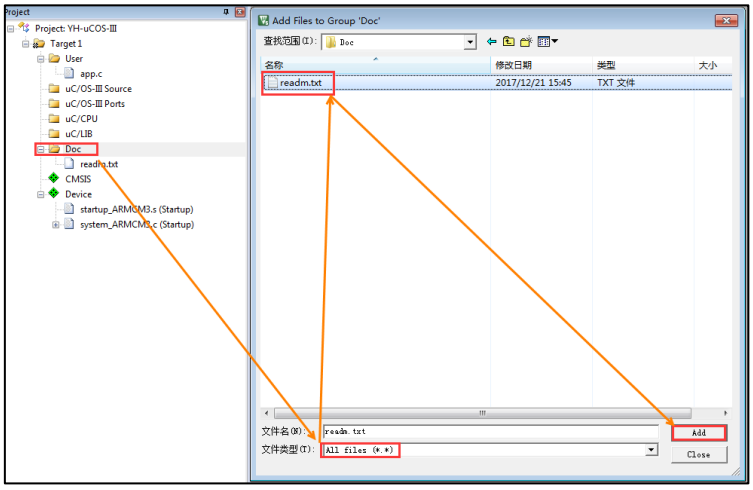


图 3-7 如何往组里面添加文件

3.4.1 编写 main 函数

一个工程如果没有 main 函数是编译不成功的，会出错。因为系统在开始执行的时候先执行启动文件里面的复位程序，复位程序里面会调用 C 库函数__main，__main 的作用是初始化好系统变量，如全局变量，只读的，可读可写的等等。__main 最后会调用__rtentry，再由__rtentry 调用 main 函数，从而由汇编跳入到 C 的世界，这里面的 main 函数就需要我们手动编写，如果没有编写 main 函数，就会出现 main 函数没有定义的错误，具体见图 3-8。

```
Build Output
*** Using Compiler 'V5.05 update 2 (build 169)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Build target 'Target 1'
compiling app.c...
linking...
.\Objects\YH-uCOS-III.axf: Error: L6218E: Undefined symbol main (referred from __rtentry2.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 1 error messages.
".\Objects\YH-uCOS-III.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:00
```

图 3-8 没定义 main 函数的错误

main 函数我们写在 app.c 文件里面，因为是刚刚新建工程，所以 main 函数暂时为空，具体见代码清单 3-1。

代码清单 3-1main 函数

```
1 int main(void)
2 {
3     for (;;) {
4         /* 啥事不干 */
5     }
6 }
```

3.5 调试配置

3.5.1 设置软件仿真

最后，我们再配置下调试相关的配置即可。为了方便，我们全部代码都用软件仿真，即不需要开发板也不需要仿真器，只需要一个 KEIL 软件即可，有关软件仿真的配置具体见图 3-9。

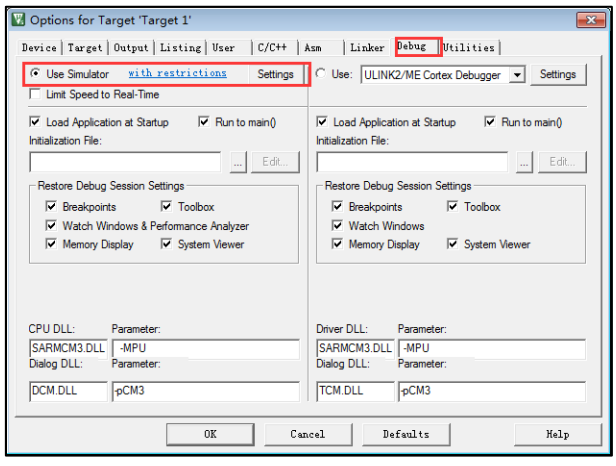


图 3-9 软件仿真的配置

3.5.2 修改时钟大小

在时钟相关文件 system_ARMCM3.c 的开头，有一段代码定义了系统时钟的大小为 25M，具体见代码清单 3-2。在软件仿真的时候，确保时间的准确性，代码里面的系统时钟

跟软件仿真的时钟必须一致，所以 Options for Target->Target 的时钟应该由默认的 12M 改成 25M，具体见图 3-10。

代码清单 3-2 时钟相关宏定义

```
1 #define __HSI             ( 8000000UL)
2 #define __XTAL            ( 5000000UL)
3
4 #define __SYSTEM_CLOCK    (5*__XTAL)
```

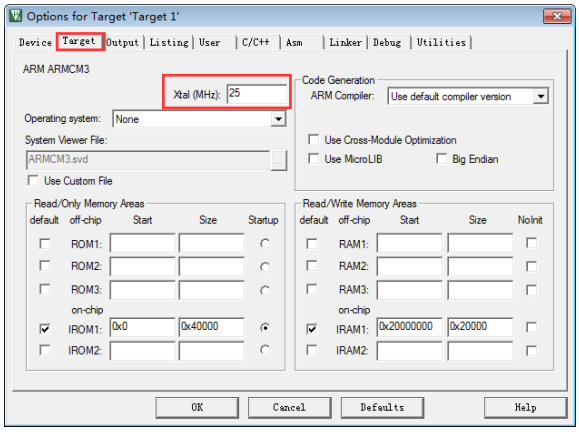


图 3-10 软件仿真时钟配置

3.5.3 添加头文件路径

在 C/C++选项卡里面指定工程头文件的路径，不然编译会出错，头文件路径的具体指定方法见图 3-11。

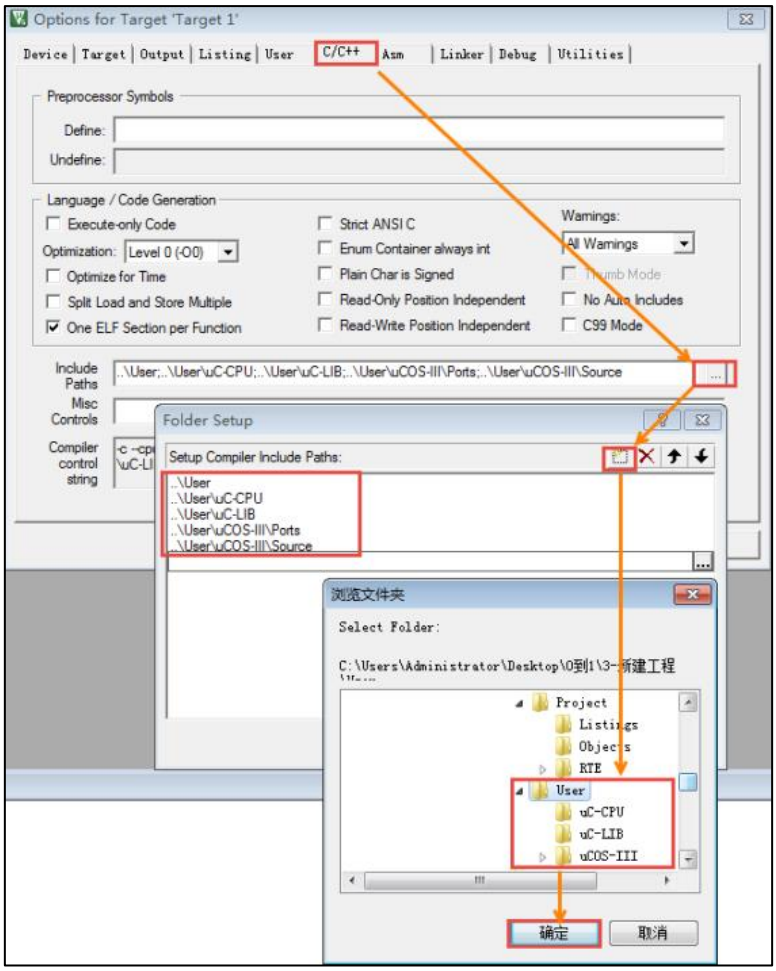


图 3-11 指定头文件的路径

至此，一个完整的基于 Cortex-M 内核的软件仿真的工程就建立完毕。

第4章　裸机系统与多任务系统

在真正开始动手写 RTOS 之前，我们先来讲解下单片机编程中的裸机系统和多任务系统的区别。

4.1 裸机系统

裸机系统通常分成轮询系统和前后台系统，有关这两者的具体实现方式请看下面的讲解。

4.1.1 轮询系统

轮询系统即是在裸机编程的时候，先初始化好相关的硬件，然后让主程序在一个死循环里面不断循环，顺序地做各种事情，大概的伪代码具体见代码清单 4-1。轮询系统是一种非常简单的软件结构，通常只适用于那些只需要顺序执行代码且不需要外部事件来驱动的就能完成的事情。在代码清单 4-1 中，如果只是实现 LED 翻转，串口输出，液晶显示等这些操作，那么使用轮询系统将会非常完美。但是，如果加入了按键操作等需要检测外部信号的事件，用来模拟紧急报警，那么整个系统的实时响应能力就不会那么好了。假设 DoSomething3 是按键扫描，当外部按键被按下，相当于一个警报，这个时候，需要立马响应，并做紧急处理，而这个时候程序刚好执行到 DoSomething1，要命的是 DoSomething1 需要执行的时间比较久，久到按键释放之后都没有执行完毕，那么当执行到 DoSomething3 的时候就会丢失掉一次事件。足见，轮询系统只适合顺序执行的功能代码，当有外部事件驱动时，实时性就会降低。

代码清单 4-1 轮询系统伪代码

```
1  int main(void)
2  {
3      /* 硬件相关初始化 */
4      HardWareInit();
5
6      /* 无限循环 */
7      for (;;) {
8          /* 处理事情 1 */
9          DoSomethin1();
10
11         /* 处理事情 2 */
12         DoSomething2();
13
14         /* 处理事情 3 */
15         DoSomething3();
16     }
17 }
```

4.1.2 前后台系统

相比轮询系统，前后台系统是在轮询系统的基础上加入了中断。外部事件的响应在中断里面完成，事件的处理还是回到轮询系统中完成，中断在这里我们称为后台，main 函数里面的无限循环我们称为后台，大概的伪代码见代码清单 4-2。

代码清单 4-2 前后台系统伪代码

```
int flag1 = 0;
int flag2 = 0;
int flag3 = 0;

int main(void)
{
    /* 硬件相关初始化 */
    HardWareInit();

    /* 无限循环 */
    for (;;) {
        if (flag1) {
            /* 处理事情 1 */
            DoSomethin1();
        }

        if (flag2) {
            /* 处理事情 2 */
            DoSomething2();
        }

        if (flag3) {
            /* 处理事情 3 */
            DoSomething3();
        }
    }
}

void ISR1(void)
{
    /* 置位标志位 */
    flag1 = 1;
    /* 如果事件处理时间很短，则在中断里面处理
       如果事件处理时间比较长，在回到前台处理 */
    DoSomethin1();
}

void ISR1(void)
{
    /* 置位标志位 */
    flag1 = 2;

    /* 如果事件处理时间很短，则在中断里面处理
       如果事件处理时间比较长，在回到前台处理 */
    DoSomethin2();
}

void ISR3(void)
{
    /* 置位标志位 */
    flag3 = 1;

    /* 如果事件处理时间很短，则在中断里面处理
```

```
54          如果事件处理时间比较长，在回到前台处理 */
55      DoSomethin3();
56  }
```

在顺序执行后台程序的时候，如果有中断来临，那么中断会打断后台程序的正常执行流，转而去执行中断服务程序，在中断服务程序里面标记事件，如果事件要处理的事情很简短，则可在中断服务程序里面处理，如果事件要处理的事情比较多，则返回到后台程序里面处理。虽然事件的响应和处理是分开了，但是事件的处理还是在后台里面顺序执行的，但相比轮询系统，前后台系统确保了事件不会丢失，再加上中断具有可嵌套的功能，这可以大大的提高程序的实时响应能力。在大多数的中小型项目中，前后台系统运用的好，堪称有操作系统的效果。

4.2 多任务系统

相比前后台系统，多任务系统的事件响应也是在中断中完成的，但是事件的处理是在任务中完成的。在多任务系统中，任务跟中断一样，也具有优先级，优先级高的任务会被优先执行。当一个紧急的事件在中断被标记之后，如果事件对应的任务的优先级足够高，就会立马得到响应。相比前后台系统，多任务系统的实时性又被提高了。多任务系统大概的伪代码具体见代码清单 4-3。

代码清单 4-3 多任务系统伪代码

```
 1  int flag1 = 0;
 2  int flag2 = 0;
 3  int flag3 = 0;
 4
 5  int main(void)
 6  {
 7      /* 硬件相关初始化 */
 8      HardWareInit();
 9
10      /* OS 初始化 */
11      RTOSInit();
12
13      /* OS 启动，开始多任务调度，不再返回 */
14      RTOSStart();
15  }
16
17  void ISR1(void)
18  {
19      /* 置位标志位 */
20      flag1 = 1;
21  }
22
23  void ISR1(void)
24  {
25      /* 置位标志位 */
26      flag1 = 2;
27  }
28
29  void ISR3(void)
30  {
31      /* 置位标志位 */
32      flag3 = 1;
33  }
34
35  void DoSomethin1(void)
36  {
```

```
37      /* 无限循环，不能返回 */
38      for (;;) {
39          /* 任务实体 */
40          if (flag1) {
41
42          }
43      }
44  }
45
46  void DoSomethin2(void)
47  {
48      /* 无限循环，不能返回 */
49      for (;;) {
50          /* 任务实体 */
51          if (flag2) {
52
53          }
54      }
55  }
56
57  void DoSomethin3(void)
58  {
59      /* 无限循环，不能返回 */
60      for (;;) {
61          /* 任务实体 */
62          if (flag3) {
63
64          }
65      }
66  }
```

相比前后台系统中后台顺序执行的程序主体，在多任务系统中，根据程序的功能，我们把这个程序主体分割成一个个独立的，无限循环且不能返回的小程序，这个小程序我们称之为任务。每个任务都是独立的，互不干扰的，且具备自身的优先级，它由操作系统调度管理。加入操作系统后，我们在编程的时候不需要精心地去设计程序的执行流，不用担心每个功能模块之间是否存在干扰。加入了操作系统，我们的编程反而变得简单了。整个系统随之带来的额外开销就是操作系统占据的那一丁点的 FLASH 和 RAM。现如今，单片机的 FLASH 和 RAM 是越来越大，完全足以抵挡 RTOS 那点开销。

无论是裸机系统中的轮询系统、前后台系统和多任务系统，我们不能一锤子的敲定孰优孰劣，它们是不同时代的产物，在各自的领域都还有相当大的应用价值，只有合适才是最好。有关这三者的软件模型区别具体见表格 4-1。

表格 4-1 轮询、前后台和多任务系统软件模型区别

模型	事件响应	事件处理	特点
轮询系统	主程序	主程序	轮询响应事件，轮询处理事件
前后台系统	中断	主程序	实时响应事件，轮询处理事件
多任务系统	中断	任务	实时响应事件，实时处理事件

第5章　任务的定义与任务切换的实现

5.1 本章目标

本章是我们真正从从 0 到 1 写 RTOS 的第一章，属于基础中的基础，必须要学会创建任务，并重点掌握任务是如何切换的。因为任务的切换是由汇编代码来完成的，所以代码看起来比较难懂，但是我会尽力把代码讲得透彻。如果本章内容学不会，后面的内容根本无从下手。

在这章中，我们会创建两个任务，并让这两个任务不断地切换，任务的主体都是让一个变量按照一定的频率翻转，通过 KEIL 的软件仿真功能，在逻辑分析仪中观察变量的波形变化，最终的波形图具体见图 5-1。

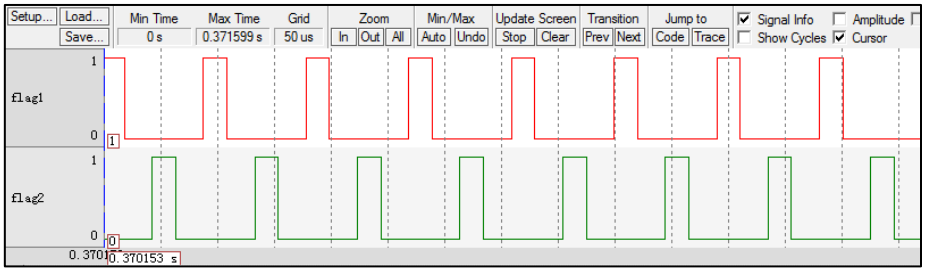


图 5-1 任务轮流切换波形图

其实，图 5-1 的波形图的效果，并不是真正的多任务系统中任务切换的效果图，这个效果其实可以完全由裸机代码来实现，具体见代码清单 5-1。

代码清单 5-1 裸机系统中两个变量轮流翻转

```
1  /* flag 必须定义成全局变量才能添加到逻辑分析仪里面观察波形
2  ** 在逻辑分析仪中要设置以 bit 的模式才能看到波形，不能用默认的模拟量
3  */
4  uint32_t flag1;
5  uint32_t flag2;
6
7
8  /* 软件延时，不必纠结具体的时间 */
9  void delay( uint32_t count )
10 {
11     for (; count!=0; count--);
12 }
13
14 int main(void)
15 {
16     /* 无限循环，顺序执行 */
17     for (;;) {
18         flag1 = 1;
19         delay( 100 );
20         flag1 = 0;
21         delay( 100 );
22
23         flag2 = 1;
24         delay( 100 );
25         flag2 = 0;
26         delay( 100 );
27     }
```

```
28 }
```

在多任务系统中，两个任务不断切换的效果图应该像图 5-2 所示那样，即两个变量的波形是完全一样的，就好像 CPU 在同时干两件事一样，这才是多任务的意义。虽然两者的波形图一样，但是，代码的实现方式是完全不一样的，由原来的顺序执行变成了任务的主动切换，这是根本区别。这章只是开始，我们先掌握好任务是如何切换，在后面章节中，我们会陆续的完善功能代码，加入系统调度，实现真正的多任务。千里之行，始于本章节，不要急。

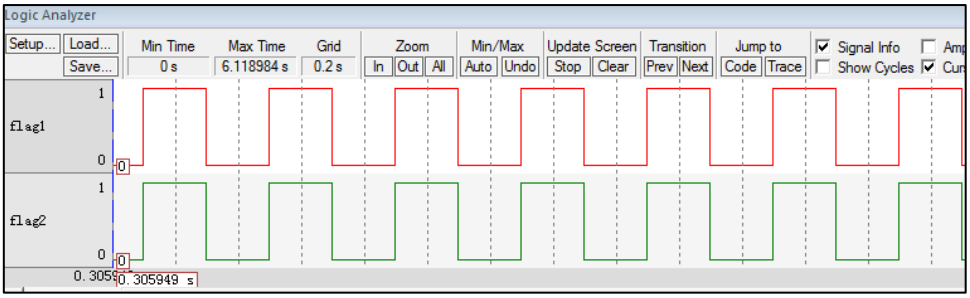


图 5-2 多任务系统任务切换波形图

5.2 什么是任务

在裸机系统中，系统的主体就是 main 函数里面顺序执行的无限循环，这个无限循环里面 CPU 按照顺序完成各种事情。在多任务系统中，我们根据功能的不同，把整个系统分割成一个个独立的且无法返回的函数，这个函数我们称为任务，也有人称之为线程。任务的大概形式具体见代码清单 5-2。

代码清单 5-2 多任务系统中任务的形式

```
1 void Task (void *parg)
2 {
3     /* 任务主体，无限循环且不能返回 */
4     for (;;) {
5         /* 任务主体代码 */
6     }
7 }
```

5.3 创建任务

5.3.1 定义任务堆栈

我们先回想下，在一个裸机系统中，如果有全局变量，有子函数调用，有中断发生。那么系统在运行的时候，全局变量放在哪里，子函数调用时，局部变量放在哪里，中断发生时，函数返回地址发哪里。如果只是单纯的裸机编程，它们放哪里我们不用管，但是如果要写一个 RTOS，这些种种环境参数，我们必须弄清楚他们是如何存储的。在裸机系统中，他们统统放在一个叫栈的地方，栈是单片机 RAM 里面一段连续的内存空间，栈的大小由启动文件里面的代码配置，具体见代码清单 5-3，最后由 C 库函数_main 进行初始化。它们在 RAM 空间里面的大概分布具体见。

代码清单 5-3 裸机系统中的栈分配

```
1 Stack_Size      EQU     0x00000400
2
3                 AREA    STACK, NOINIT, READWRITE, ALIGN=3
4 Stack_Mem       SPACE   Stack_Size
5 __initial_sp
```

但是，在多任务系统中，每个任务都是独立的，互不干扰的，所以要为每个任务都分配独立的栈空间，这个栈空间通常是一个预先定义好的全局数组。这些一个个的任务栈也是存在于 RAM 中，能够使用的最大的栈也是由代码清单 5-3 中的 Stack_Size 决定。 只是多任务系统中任务的堆栈就是在统一的一个栈空间里面分配好一个个独立的房间，每个任务只能使用各自的房间，而裸机系统中需要使用栈的时候则可以天马行空，随便在栈里面找个空闲的空间使用，大概的区别具体见。

本章我们要实现两个变量按照一定的频率轮流的翻转，需要两个任务来实现，那么就需要定义两个任务栈，具体见代码清单 5-4。在多任务系统中，有多少个任务就需要定义多少个任务堆栈。

代码清单 5-4 定义任务堆栈

```
1 #define  TASK1_STK_SIZE        128             (1)
2 #define  TASK2_STK_SIZE        128
3
4 static   CPU_STK   Task1Stk[TASK1_STK_SIZE];   (2)
5 static   CPU_STK   Task2Stk[TASK2_STK_SIZE];
```

代码清单 5-4 （1）任务栈的大小由宏定义控制，在 uC/OS-III 中，空闲任务的堆栈最小应该大于 128，那么我们这里的任务的堆栈也暂且配置为 128。

代码清单 5-4 （2）任务栈其实就是一个预先定义好的全局数据，数据类型为 CPU_STK。在 uC/OS-III 中，凡是涉及到数据类型的地方，uC/OS-II 都会将标准的 C 数据类型用 typedef 重新取一个类型名，命名方式则采用见名之义的方式命名且统统大写。凡是与 CPU 类型相关的数据类型则统一在 cpu.h 中定义，与 OS 相关的数据类型则在 os_type.h 定义。CPU_STK 就是与 CPU 相关的数据类型，则在 cpu.h 中定义，具体见代码清单 5-5。cpu.h 首次使用则需要自行在 uC-CPU 文件夹中新建并添加到工程的 uC/CPU 这个组中。代码清单 5-5 中除了 CPU_STK 外，其它数据类型重定义是本章后面内容需要使用到，这里统一贴出来，后面将不再赘述。

代码清单 5-5 cpu.h 中的数据类型

```
1 #ifndef CPU_H
2 #define CPU_H
3
4 typedef  unsigned  short       CPU_INT16U;
5 typedef  unsigned  int         CPU_INT32U;
6 typedef  unsigned  char        CPU_INT08U;
7
8 typedef  CPU_INT32U  CPU_ADDR;
9
10 /* 堆栈数据类型重定义 */
11 typedef  CPU_INT32U             CPU_STK;
12 typedef  CPU_ADDR               CPU_STK_SIZE;
13
14 typedef  volatile  CPU_INT32U  CPU_REG32;
15
```

```
16 #endif /* CPU_H */
```

5.3.2 定义任务函数

任务是一个独立的函数，函数主体无限循环且不能返回。本章我们定义的两个任务具体见代码清单 5-6。

代码清单 5-6 任务函数

```
 1 /* flag 必须定义成全局变量才能添加到逻辑分析仪里面观察波形
 2 ** 在逻辑分析仪中要设置以 bit 的模式才能看到波形，不能用默认的模拟量
 3 */
 4 uint32_t flag1;                          (1)
 5 uint32_t flag2;
 6
 7
 8 /* 任务 1 */
 9 void Task1( void *p_arg )                 (2)
10 {
11     for ( ;; ) {
12         flag1 = 1;
13         delay( 100 );
14         flag1 = 0;
15         delay( 100 );
16     }
17 }
18
19 /* 任务 2 */
20 void Task2( void *p_arg )                 (3)
21 {
22     for ( ;; ) {
23         flag2 = 1;
24         delay( 100 );
25         flag2 = 0;
26         delay( 100 );
27     }
28 }
```

代码清单 5-6 （1）：需要在 KEIL 的逻辑分析仪中观察波形的变量需要定义成全局变量，且要以 bit 的模式观察，不能使用默认的模拟量。

代码清单 5-6 （2）和（3）：正如我们所说的那样，任务是一个独立的、无限循环且不能返回的函数。

5.3.3 定义任务控制块 TCB

在裸机系统中，程序的主体是 CPU 按照顺序执行的。而在多任务系统中，任务的执行是由系统调度的。系统为了顺利的调度任务，为每个任务都额外定义了一个任务控制块 TCB（Task Control Block），这个任务控制块就相当于任务的身份证，里面存有任务的所有信息，比如任务的堆栈，任务名称，任务的形参等。有了这个任务控制块之后，以后系统对任务的全部操作都可以通过这个 TCB 来实现。TCB 是一个新的数据类型，在 os.h（os.h 第一次使用需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III

Source 组）这个头文件中声明，有关 TCB 具体的声明见代码清单 5-7，使用它可以为每个任务都定义一个 TCB 实体。

代码清单 5-7 任务控制块 TCB 类型声明

```
1 /* 任务控制块重定义 */
2 typedef  struct  os_tcb              OS_TCB;          (1)
3
4 /* 任务控制块 数据类型声明 */
5 struct os_tcb {                                      (2)
6     CPU_STK          *StkPtr;
7     CPU_STK_SIZE     StkSize;
8 };
```

代码清单 5-7 （1）：在 uC/OS-III 中，所有的数据类型都会重新取一个名字且用大写字母表示。

代码清单 5-7 （2）：目前 TCB 里面的成员还比较少，只有堆栈指针和堆栈大小。其中为了以后操作方便，我们把堆栈指针作为 TCB 的第一个成员。

在本章实验中，我们在 app.c 文件中为两个任务定义的 TCB 具体见代码清单 5-8。

代码清单 5-8 任务 TCB 定义

```
1 static   OS_TCB    Task1TCB;
2 static   OS_TCB    Task2TCB;
```

5.3.4 实现任务创建函数

任务的堆栈，任务的函数实体，任务的 TCB 最终需要联系起来才能由系统进行统一调度。那么这个联系的工作就由任务创建函数 OSTaskCreate 来实现，该函数在 os_task.c（os_task.c 第一次使用需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III Source 组）中定义，所有跟任务相关的函数都在这个文件定义。OSTaskCreate 函数的实现具体见代码清单 5-9。

代码清单 5-9 OSTaskCreate 函数

```
1 void OSTaskCreate (OS_TCB        *p_tcb,             (1)
2                    OS_TASK_PTR    p_task,            (2)
3                    void          *p_arg,             (3)
4                    CPU_STK       *p_stk_base,        (4)
5                    CPU_STK_SIZE  stk_size,           (5)
6                    OS_ERR        *p_err)             (6)
7 {
8     CPU_STK       *p_sp;
9
10    p_sp = OSTaskStkInit (p_task,                    (7)
11                          p_arg,
12                          p_stk_base,
13                          stk_size);
14    p_tcb->StkPtr = p_sp;                            (8)
15    p_tcb->StkSize = stk_size;                       (9)
16
17    *p_err = OS_ERR_NONE;                            (10)
18 }
```

代码清单 5-9： OSTaskCreate 函数遵循 uC/OS-III 中的函数命名规则，以大小的 OS 开头，表示这是一个外部函数，可以由用户调用，以 OS_开头的函数表示内部函数，只能由 uC/OS-III 内部使用。紧接着是文件名，表示该函数放在哪个文件，最后是函数功能名称。

代码清单 5-9 （1）：p_tcb 是任务控制块指针。

代码清单 5-9 （2）：p_task 是任务函数名，类型为 OS_TASK_PTR，原型声明在 os.h 中，具体见代码清单 5-10。

代码清单 5-10 OS_TASK_PTR 原型声明

```
1 typedef  void                          (*OS_TASK_PTR)(void *p_arg);
```

代码清单 5-9 （3）：p_arg 是任务形参，用于传递任务参数。

代码清单 5-9 （4）：p_stk_base 用于指向任务堆栈的起始地址。

代码清单 5-9 （5）：stk_size 表示任务堆栈的大小。

代码清单 5-9 （6）：p_err 用于存错误码，uC/OS-III 中为函数的返回值预先定义了很多错误码，通过这些错误码我们可以知道函数是因为什么出错。为了方便，我们现在把 uC/OS-III 中所有的错误号都贴出来，错误码是枚举类型的数据，在 os.h 中定义，具体见代码清单 5-11。

代码清单 5-11 错误码枚举定义

```
1 typedef  enum  os_err {
2     OS_ERR_NONE                       =     0u,
3
4     OS_ERR_A                          = 10000u,
5     OS_ERR_ACCEPT_ISR                 = 10001u,
6
7     OS_ERR_B                          = 11000u,
8
9     OS_ERR_C                          = 12000u,
10    OS_ERR_CREATE_ISR                 = 12001u,
11
12    /* 篇幅限制，中间部分删除，具体的可查看本章配套的例程 */
199
200    OS_ERR_X                          = 33000u,
201
202    OS_ERR_Y                          = 34000u,
203    OS_ERR_YIELD_ISR                  = 34001u,
204
205    OS_ERR_Z                          = 35000u
206 } OS_ERR;
```

代码清单 5-9 （7）：OSTaskStkInit()是任务堆栈初始化函数。当任务第一次运行的时候，加载到 CPU 寄存器的参数就放在任务堆栈里面，在任务创建的时候，预先初始化好堆栈。OSTaskStkInit()函数在 os_cpu_c.c（os_cpu_c.c 第一次使用需要自行在文件夹 uC-CPU 中新建并添加到工程的 uC/CPU 组）中定义，具体见代码清单 5-12。

代码清单 5-12 OSTaskStkInit()函数

```
1 CPU_STK *OSTaskStkInit (OS_TASK_PTR  p_task,                    (1)
2                         void         *p_arg,                    (2)
3                         CPU_STK      *p_stk_base,               (3)
4                         CPU_STK_SIZE stk_size)                  (4)
5 {
6     CPU_STK  *p_stk;
7
8     p_stk = &p_stk_base[stk_size];                              (5)
9     /* 异常发生时自动保存的寄存器 */                            (6)
10    *--p_stk = (CPU_STK)0x01000000u;    /* xPSR 的 bit24 必须置 1  */
11    *--p_stk = (CPU_STK)p_task;         /* R15(PC)任务的入口地址  */
```

```
12      *--p_stk = (CPU_STK)0x14141414u;    /* R14 (LR)              */
13      *--p_stk = (CPU_STK)0x12121212u;    /* R12                   */
14      *--p_stk = (CPU_STK)0x03030303u;    /* R3                    */
15      *--p_stk = (CPU_STK)0x02020202u;    /* R2                    */
16      *--p_stk = (CPU_STK)0x01010101u;    /* R1                    */
17      *--p_stk = (CPU_STK)p_arg;          /* R0 : 任务形参          */
18      /* 异常发生时需手动保存的寄存器 */                            (7)
19      *--p_stk = (CPU_STK)0x11111111u;    /* R11                   */
20      *--p_stk = (CPU_STK)0x10101010u;    /* R10                   */
21      *--p_stk = (CPU_STK)0x09090909u;    /* R9                    */
22      *--p_stk = (CPU_STK)0x08080808u;    /* R8                    */
23      *--p_stk = (CPU_STK)0x07070707u;    /* R7                    */
24      *--p_stk = (CPU_STK)0x06060606u;    /* R6                    */
25      *--p_stk = (CPU_STK)0x05050505u;    /* R5                    */
26      *--p_stk = (CPU_STK)0x04040404u;    /* R4                    */
27
28      return (p_stk);                                               (8)
29  }
```

代码清单 5-12（1）：p_task 是任务名，指示着任务的入口地址，在任务切换的时候，需要加载到 R15，即 PC 寄存器，这样 CPU 就可以找到要运行的任务。

代码清单 5-12（2）：p_arg 是任务的形参，用于传递参数，在任务切换的时候，需要加载到寄存器 R0。R0 寄存器通常用来传递参数。

代码清单 5-12（3）：p_stk_base 表示任务堆栈的起始地址。

代码清单 5-12（4）：stk_size 表示任务堆栈的大小，数据类型为 CPU_STK_SIZE，在 Cortex-M3 内核的处理器中等于 4 个字节，即一个字。

代码清单 5-12（5）：获取任务堆栈的栈顶地址，ARMCM3 处理器的栈是由高地址向低地址生长的。所以初始化栈之前，要获取到栈顶地址，然后栈地址逐一递减即可。

代码清单 5-12（6）：任务第一次运行的时候，加载到 CPU 寄存器的环境参数我们要预先初始化好。初始化的顺序固定，首先是异常发生时自动保存的 8 个寄存器，即 xPSR、R15、R14、R12、R3、R2、R1 和 R0。其中 xPSR 寄存器的位 24 必须是 1，R15 PC 指针必须存的是任务的入口地址，R0 必须是任务形参，剩下的 R14、R12、R3、R2 和 R1 为了调试方便，填入与寄存器号相对应的 16 进制数。

代码清单 5-12（7）：剩下的是 8 个需要手动加载到 CPU 寄存器的参数，为了调试方便填入与寄存器号相对应的 16 进制数。

代码清单 5-12（8）：返回栈指针 p_stk，这个时候 p_stk 指向剩余栈的栈顶。

代码清单 5-9 （8）：将剩余栈的栈顶指针 p_sp 保存到任务控制块 TCB 的第一个成员 StkPtr 中。

代码清单 5-9 （9）：将任务堆栈的大小保存到任务控制块 TCB 的成员 StkSize 中。

代码清单 5-9 （10）：函数执行到这里表示没有错误，即 OS_ERR_NONE。

任务创建好之后，我们需要把任务添加到一个叫就绪列表的数组里面，表示任务已经就绪，系统随时可以调度。将任务添加到就绪列表的代码具体见代码清单 5-13。

代码清单 5-13 将任务添加到就绪列表

```
1 /* 将任务加入到就绪列表 */
2 OSRdyList[0].HeadPtr = &Task1TCB;          (1)
3 OSRdyList[1].HeadPtr = &Task2TCB;          (2)
```

代码清单 5-13 （1）和（2）：把任务 TCB 指针放到 OSRDYList 数组里面。OSRDYList 是一个类型为 OS_RDY_LIST 的全局变量，在 os.h 中定义，具体见代码清单 5-14。

代码清单 5-14 全局变量 OSRDYList 定义

```
      (1)      (2)                              (3)
1 OS_EXT     OS_RDY_LIST     OSRdyList[OS_CFG_PRIO_MAX];
```

代码清单 5-14 （3）：OS_CFG_PRIO_MAX 是一个定义，表示这个系统支持多少个优先级（刚开始暂时不支持多个优先级，往后章节会支持），目前这里仅用来表示这个就绪列表可以存多少个任务的 TCB 指针。具体的宏在 os_cfg.h（os_cfg.h 第一次使用需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III Source 组）中定义，具体见代码清单 5-15。

代码清单 5-15 OS_CFG_PRIO_MAX 宏定义

```
1 #ifndef OS_CFG_H
2 #define OS_CFG_H
3
4 /* 支持最大的优先级 */
5 #define OS_CFG_PRIO_MAX                 32u
6
7
8 #endif /* OS_CFG_H */
```

代码清单 5-14 （2）：OS_RDY_LIST 是就绪列表的数据类型，在 os.h 中声明，具体见代码清单 5-16。

代码清单 5-16 OS_RDY_LIST 数据类型声明

```
1 typedef  struct  os_rdy_list          OS_RDY_LIST;          (1)
2
3 struct os_rdy_list {                                        (2)
4     OS_TCB         *HeadPtr;
5     OS_TCB         *TailPtr;
6 };
```

代码清单 5-16 （1）：uC/OS-III 中 中会为每个数据类型重新取一个大写的名字。

代码清单 5-16 （2）：OS_RDY_LIST 里面目前暂时只有两个 TCB 类型的指针，一个是头指针，一个是尾指针。本章实验只用到头指针，用来指向任务的 TCB。只有当后面讲到同一个优先级支持多个任务的时候才需要使用头尾指针来将 TCB 串成一个双向链表。

代码清单 5-14（1）：OS_EXTI 是一个在 os.h 中定义的宏，具体见代码清单 5-17。

代码清单 5-17 OS_EXTI 宏定义

```
1 #ifdef      OS_GLOBALS
2 #define     OS_EXT
3 #else
4 #define     OS_EXT  extern
5 #endif
```

代码清单 5-17：该段代码的意思是，如果没有定义 OS_GLOBALS 这个宏，那么 OS_EXTI 就为空，否则就为 extern。

在 uC/OS-III 中，需要使用很多全局变量，这些全局变量都在 os.h 这个头文件中定义，但是 os.h 会被包含进很多的文件中，那么编译的时候，os.h 里面定义的全局变量就会出现

重复定义的情况，而我们要的只是 os.h 里面定义的全局变量只定义一次，其它包含 os.h 头文件的时候只是声明。有人说，那我可以加 extern，那你告诉我怎么加？

通常我们的做法都是在 C 文件里面定义全局变量，然后在头文件里面加 extern 声明，哪里需要使用就在哪里加 extern 声明。但是 uC/OS-III 中，文件非常多，这种方法可行，但不现实。所以就有了现在在 os.h 头文件中定义全局变量，然后在 os.h 文件的开头加上代码清单 5-17 的宏定义的方法。但是到了这里还没成功，uC/OS-III 再另外新建了一个 os_var.c（os_aar.c 第一次使用需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III Source 组）的文件，在里面包含 os.h，且只在这个文件里面定义 OS_GLOBALS 这个宏，具体见代码清单 5-18。

代码清单 5-18 os_var.c 文件内容

```
1 #define   OS_GLOBALS
2
3 #include "os.h"
```

经过这样处理之后，在编译整个工程的时候，只有 var.c 里面的 os.h 的 OS_EXTI 才会被替换为空，即变量的定义，其它包含 os.h 的文件因为没有定义 OS_GLOBAS 这个宏，则 OS_EXTI 会被替换成 extern，即变成了变量的声明。这样就实现了在头文件中定义变量。

在 uC/OS-III 中，将任务添加到就绪列表其实是在 OSTaskCreate()函数中完成的。每当任务创建好就把任务添加到就绪列表，表示任务已经就绪。只是目前这里的就绪列表的实现还是比较简单，不支持优先级，不支持双向链表，只是简单的将任务控制块放到就绪列表的数组里面。后面会有独立的章节来讲解就绪列表，等我们完善就绪列表之后，再把这部分的操作放回 OSTaskCreate()函数里面。

5.4 OS 系统初始化

OS 系统初始化一般是在硬件初始化完成之后来做的，主要做的工作就是初始化 uC/OS-III 中定义的全局变量。OSInit()函数在文件 os_core.c（os_core.c 第一次使用需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III Source 组）中定义，具体实现见代码清单 5-19。

代码清单 5-19 OSInit()函数

```
1 void OSInit (OS_ERR *p_err)
 2 {
 3     OSRunning =  OS_STATE_OS_STOPPED;              (1)
 4
 5     OSTCBCurPtr = (OS_TCB *)0;                     (2)
 6     OSTCBHighRdyPtr = (OS_TCB *)0;                 (3)
 7
 8     OS_RdyListInit();                              (4)
 9
10     *p_err = OS_ERR_NONE;                          (5)
11 }
```

代码清单 5-19（1）：系统用一个全局变量 OSRunning 来指示系统的运行状态，刚开始系统初始化的时候，默认为停止状态，即 OS_STATE_OS_STOPPED。

代码清单 5-19（2）：全局变量 OSTCBCurPtr 是系统用于指向当前正在运行的任务的 TCB 指针，在任务切换的时候用得到。

代码清单 5-19（3）：全局变量 OSTCBHighRdyPtr 用于指向就绪任务中优先级最高的任务的 TCB，在任务切换的时候用得到。本章暂时不支持优先级，则用于指向第一个运行的任务的 TCB。

代码清单 5-19（4）：OS_RdyListInit()用于初始化全局变量 OSRdyList[]，即初始化就绪列表。OS_RdyListInit()在 os_core.c 文件中定义，具体实现见代码清单 5-20

代码清单 5-20 OS_RdyListInit() 函数

```
1  void OS_RdyListInit(void)
2  {
3      OS_PRIO i;
4      OS_RDY_LIST *p_rdy_list;
5
6      for ( i=0u; i<OS_CFG_PRIO_MAX; i++ ) {
7          p_rdy_list = &OSRdyList[i];
8          p_rdy_list->HeadPtr = (OS_TCB *)0;
9          p_rdy_list->TailPtr = (OS_TCB *)0;
10      }
11 }
```

代码清单 5-19（5）：代码运行到这里表示没有错误，即 OS_ERR_NONE。

代码清单 5-19 中的全局变量 OSTCBCurPtr 和 OSTCBHighRdyPtr 均在 os.h 中定义，具体见代码清单 5-21。OS_STATE_OS_STOPPED 这个表示系统运行状态的宏也在 os.h 中定义，具体见代码清单 5-22。

代码清单 5-21 OSInit()函数中出现的全局变量的定义

```
1 OS_EXT     OS_TCB          *OSTCBCurPtr;
2 OS_EXT     OS_TCB          *OSTCBHighRdyPtr;
3 OS_EXT     OS_RDY_LIST     OSRdyList[OS_CFG_PRIO_MAX];
4 OS_EXT     OS_STATE        OSRunning;
```

代码清单 5-22 系统状态的宏定义

```
1 #define  OS_STATE_OS_STOPPED                    (OS_STATE)(0u)
2 #define  OS_STATE_OS_RUNNING                    (OS_STATE)(1u)
```

5.5 启动系统

任务创建好，系统初始化完毕之后，就可以开始启动系统了。系统启动函数 OSStart()在 os_core.c 中定义，具体实现见代码清单 5-23。

代码清单 5-23 OSStart() 函数

```
1  void OSStart (OS_ERR *p_err)
2  {
3      if ( OSRunning == OS_STATE_OS_STOPPED ) {           (1)
4          /* 手动配置任务 1 先运行 */
5          OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;         (2)
6
7          /* 启动任务切换，不会返回 */
8          OSStartHighRdy();                               (3)
9
```

```
10          /* 不会运行到这里，运行到这里表示发生了致命的错误 */
11          *p_err = OS_ERR_FATAL_RETURN;
12      } else {
13          *p_err = OS_STATE_OS_RUNNING;
14      }
15  }
```

代码清单 5-23（1）：系统是第一次启动的话，if 肯定为真，则继续往下运行。

代码清单 5-23（2）：OSTCBHIghRdyPtr 指向第一个要运行的任务的 TCB。因为暂时不支持优先级，所以系统启动时先手动指定第一个要运行的任务。

代码清单 5-23（3）：OSStartHighRdy()用于启动任务切换，即配置 PendSV 的优先级为最低，然后触发 PendSV 异常，在 PendSV 异常服务函数中进行任务切换。该函数不再返回，在文件 os_cpu_a.s（os_cpu_a.s 第一次使用需要自行在文件夹 uCOS-III\Ports 中新建并添加到工程的 uC/OS-III Ports 组）中定义，由汇编语言编写，具体实现见代码清单 5-24。os_cpu_a.s 文件中涉及到的 ARM 汇编指令的用法具体见表格 5-1。

表格 5-1 常用的 ARM 汇编指令讲解

指令名称	作用
EQU	给数字常量取一个符号名，相当于 C 语言中的 define
AREA	汇编一个新的代码段或者数据段
SPACE	分配内存空间
PRESERVE8	当前文件堆栈需按照 8 字节对齐
EXPORT	声明一个标号具有全局属性，可被外部的文件使用
DCD	以字为单位分配内存，要求 4 字节对齐，并要求初始化这些内存
PROC	定义子程序，与 ENDP 成对使用，表示子程序结束
WEAK	弱定义，如果外部文件声明了一个标号，则优先使用外部文件定义的标号，如果外部文件没有定义也不出错。要注意的是：这个不是 ARM 的指令，是编译器的，这里放在一起只是为了方便。
IMPORT	声明标号来自外部文件，跟 C 语言中的 EXTERN 关键字类似
B	跳转到一个标号
ALIGN	编译器对指令或者数据的存放地址进行对齐，一般需要跟一个立即数，缺省表示 4 字节对齐。要注意的是：这个不是 ARM 的指令，是编译器的，这里放在一起只是为了方便。
END	到达文件的末尾，文件结束
IF,ELSE,ENDIF	汇编条件分支语句，跟 C 语言的 if else 类似

代码清单 5-24 OSStartHighRdy() 函数

```
1  ;*********************************************************************
2  ;                                开始第一次上下文切换
3  ; 1、配置 PendSV 异常的优先级为最低
4  ; 2、在开始第一次上下文切换之前，设置 psp=0
5  ; 3、触发 PendSV 异常，开始上下文切换
6  ;*********************************************************************
7  OSStartHighRdy
8    LDR      R0, = NVIC_SYSPRI14     ; 设置  PendSV 异常优先级为最低        (1)
9    LDR      R1, = NVIC_PENDSV_PRI
10   STRB     R1, [R0]
11
12   MOVS     R0, #0                  ;设置 psp 的值为 0，开始第一次上下文切换 (2)
13   MSR      PSP, R0
14
15   LDR      R0, =NVIC_INT_CTRL      ; 触发 PendSV 异常                     (3)
```

```
16    LDR     R1, =NVIC_PENDSVSET
17    STR     R1, [R0]
18
19    CPSIE   I                       ; 使能总中断，NMI 和 HardFault 除外     (4)
20
21  OSStartHang
22    B       OSStartHang             ; 程序应永远不会运行到这里
```

代码清单 5-24 中涉及到的 NVIC_INT_CTRL、NVIC_SYSPRI14、NVIC_PENDSV_PRI 和 NVIC_PENDSVSET 这四个常量在 os_cpu_a.s 的开头定义，具体见代码清单 5-25，有关这四个常量的含义看代码注释即可。

代码清单 5-25 NVIC_INT_CTRL、NVIC_SYSPRI14、NVIC_PENDSV_PRI 和 NVIC_PENDSVSET 常量定义

```
1  ;*******************************************************************
2  ;                                常量
3  ;*******************************************************************
4  ;-------------------------------------------------------------------
5  ;有关内核外设寄存器定义可参考官方文档：STM32F10xxx Cortex-M3 programming manual
6  ;系统控制块外设 SCB 地址范围：0xE000ED00-0xE000ED3F
7  ;-------------------------------------------------------------------
8  NVIC_INT_CTRL    EQU     0xE000ED04     ; 中断控制及状态寄存器 SCB_ICSR。
9  NVIC_SYSPRI14    EQU     0xE000ED22     ; 系统优先级寄存器 SCB_SHPR3：
10                                         ; bit16~23
11 NVIC_PENDSV_PRI  EQU           0xFF     ; PendSV 优先级的值(最低)。
12 NVIC_PENDSVSET   EQU     0x10000000     ; 触发 PendSV 异常的值 Bit28：PENDSVSET
```

代码清单 5-24（1）：配置 PendSV 的优先级为 0XFF，即最低。在 uC/OS-III 中，上下文切换是在 PendSV 异常服务程序中执行的，配置 PendSV 的优先级为最低，从而消灭了在中断服务程序中执行上下文切换的可能。

代码清单 5-24（2）：设置 PSP 的值为 0，开始第一个任务切换。在任务中，使用的栈指针都是 PSP，后面如果判断出 PSP 为 0，则表示第一次任务切换。

代码清单 5-24（3）：触发 PendSV 异常，如果中断使能且有编写 PendSV 异常服务函数的话，则内核会响应 PendSV 异常，去执行 PendSV 异常服务函数。

代码清单 5-24（4）：开中断，因为有些用户在 main 函数开始会先关掉中断，等全部初始化完成后，在启动 OS 的时候才开中断。为了快速地开关中断， CM3 专门设置了一条 CPS 指令，有 4 种用法，具体见代码清单 5-26。

代码清单 5-26 CPS 指令用法

```
1 CPSID I ;PRIMASK=1      ;关中断
2 CPSIE I ;PRIMASK=0      ;开中断
3 CPSID F ;FAULTMASK=1    ;关异常
4 CPSIE F ;FAULTMASK=0    ;开异常
```

代码清单 5-26 中 PRIMASK 和 FAULTMAST 是 CM3 里面三个中断屏蔽寄存器中的两个，还有一个是 BASEPRI，有关这三个寄存器的详细用法见表格 5-2。

表格 5-2 CM3 中断屏蔽寄存器组描述

名字	功能描述
PRIMASK	这是个只有单一比特的寄存器。 在它被置 1 后，就关掉所有可屏蔽的异常，只剩下 NMI 和硬 FAULT 可以响应。它的缺省值是 0，表示没有关中断。
FAULTMASK	这是个只有 1 个位的寄存器。当它置 1 时，只有 NMI 才能响应，所有其它的异常，甚至是硬 FAULT，也通通闭嘴。它的缺省值也是 0，表示没有关异常。
BASEPRI	这个寄存器最多有 9 位（ 由表达优先级的位数决定）。它定义了被屏蔽优先

	级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断， 0 也是缺省值。

5.6 任务切换

当调用 OSStartHighRdy()函数，触发 PendSV 异常后，就需要编写 PendSV 异常服务函数，然后在里面进行任务切换。PendSV 异常服务函数具体见代码清单 5-27。PendSV 异常服务函数名称必须与启动文件里面向量表中 PendSV 的向量名一致，如果不一致则内核是响应不了用户编写的 PendSV 异常服务函数的，只响应启动文件里面默认的 PendSV 异常服务函数。启动文件里面为每个异常都编写好默认的异常服务函数，函数体都是一个死循环，当你发现代码跳转到这些启动文件里面默认的异常服务函数的时候，就要检查下异常函数名称是否写错了，没有跟向量表里面的一致。PendSV_Handler 函数里面涉及到的 ARM 汇编指令的讲解具体见表格 5-3。

代码清单 5-27 PendSV 异常服务函数

```
1  ;*********************************************************************
2  ;                                 PendSVHandler 异常
3  ;*********************************************************************
4  PendSV_Handler
5  ; 关中断，NMI 和 HardFault 除外，防止上下文切换被中断
6    CPSID   I                                                    (1)
7
8  ; 将 psp 的值加载到 R0
9    MRS     R0, PSP                                              (2)
10
11 ; 判断 R0，如果值为 0 则跳转到 OS_CPU_PendSVHandler_nosave
12 ; 进行第一次任务切换的时候，R0 肯定为 0
13   CBZ     R0, OS_CPU_PendSVHandler_nosave                      (3)
14
15 ;-----------------------一、保存上文-----------------------------
16 ; 任务的切换，即把下一个要运行的任务的堆栈内容加载到 CPU 寄存器中
17 ;----------------------------------------------------------------
18 ; 在进入 PendSV 异常的时候，当前 CPU 的 xPSR，PC（任务入口地址），
19 ; R14，R12，R3，R2，R1，R0 会自动存储到当前任务堆栈，
20 ; 同时递减 PSP 的值，随便通过 代码：MRS R0, PSP 把 PSP 的值传给 R0
21
22 ; 手动存储 CPU 寄存器 R4-R11 的值到当前任务的堆栈
23   STMDB   R0!, {R4-R11}                                        (15)
24
25
26 ; 加载 OSTCBCurPtr 指针的地址到 R1，这里 LDR 属于伪指令
27   LDR     R1, = OSTCBCurPtr                                    (16)
28 ; 加载 OSTCBCurPtr 指针到 R1，这里 LDR 属于 ARM 指令
29   LDR     R1, [R1]                                             (17)
30 ; 存储 R0 的值到 OSTCBCurPtr->OSTCBStkPtr，这个时候 R0 存的是任务空闲栈的栈顶
31   STR     R0, [R1]                                             (18)
32
33 ;-----------------------二、切换下文-----------------------------
34 ; 实现 OSTCBCurPtr = OSTCBHighRdyPtr
35 ; 把下一个要运行的任务的堆栈内容加载到 CPU 寄存器中
36 ;----------------------------------------------------------------
37 OS_CPU_PendSVHandler_nosave                                     (4)
38
39 ; 加载 OSTCBCurPtr 指针的地址到 R0，这里 LDR 属于伪指令
```

```
40    LDR      R0, = OSTCBCurPtr                                    (5)
41  ; 加载 OSTCBHighRdyPtr 指针的地址到 R1，这里 LDR 属于伪指令
42    LDR      R1, = OSTCBHighRdyPtr                                (6)
43  ; 加载 OSTCBHighRdyPtr 指针到 R2，这里 LDR 属于 ARM 指令
44    LDR      R2, [R1]                                             (7)
45  ; 存储 OSTCBHighRdyPtr 到 OSTCBCurPtr
46    STR      R2, [R0]                                             (8)
47
48  ; 加载 OSTCBHighRdyPtr 到 R0
49    LDR      R0, [R2]                                             (9)
50  ; 加载需要手动保存的信息到 CPU 寄存器 R4-R11
51    LDMIA    R0!, {R4-R11}                                        (10)
52
53  ; 更新 PSP 的值，这个时候 PSP 指向下一个要执行的任务的堆栈的栈底
54  ;（这个栈底已经加上刚刚手动加载到 CPU 寄存器 R4-R11 的偏移）
55    MSR      PSP, R0                                              (11)
56
57  ; 确保异常返回使用的堆栈指针是 PSP，即 LR 寄存器的位 2 要为 1
58    ORR      LR, LR, #0x04                                        (12)
59
60  ; 开中断
61    CPSIE    I                                                    (13)
62
63  ; 异常返回，这个时候任务堆栈中的剩下内容将会自动加载到 xPSR，
64  ; PC（任务入口地址），R14，R12，R3，R2，R1，R0（任务的形参）
65  ; 同时 PSP 的值也将更新，即指向任务堆栈的栈顶。
66  ; 在 STM32 中，堆栈是由高地址向低地址生长的。
67    BX       LR                                                   (14)
```

代码清单 5-27 PendSV 异常服务中主要完成两个工作，一是保存上文，即保存当前正在运行的任务的环境参数；二是切换下文，即把下一个需要运行的任务的环境参数从任务堆栈中加载到 CPU 寄存器，从而实现任务的切换。 接下来具体讲解下代码清单 5-27 每句代码的含义。

代码清单 5-27 PendSV 异常服务中 用到了 OSTCBCurPtr 和 OSTCBHighRdyPtr 这两个全局变量，这两个全局变量在 os.h 中定义，要想在汇编文件 os_cpu_a.s 中使用，必须将这两个全局变量导入到 os_cpu_a.s 中，具体如何导入见代码清单 5-28。

代码清单 5-28 导入 OSTCBCurPtr 和 OSTCBHighRdyPtr 到 os_cpu_a.s

```
1  ;********************************************************************
2  ;                                全局变量&函数
3  ;********************************************************************
4      IMPORT  OSTCBCurPtr                    ; 外部文件引人的参考          (1)
5      IMPORT  OSTCBHighRdyPtr
6
7      EXPORT  OSStartHighRdy                 ; 该文件定义的函数            (2)
8      EXPORT  PendSV_Handler
```

代码清单 5-28 （1）：使用 IMPORT 关键字将 os.h 中的 OSTCBCurPtr 和 OSTCBHighRdyPtr 这两个全局变量导入到该汇编文件，从而该汇编文件可以使用这两个变量。如果是函数也可以使用 IMPORT 导入的方法。

代码清单 5-28 （2）：使用 EXPORT 关键字导出该汇编文件里面的 OSStartHighRdy 和 PendSV_Handler 这两个函数，让外部文件可见。除了使用 EXPORT 导出外，还要在某个 C 的头文件里面声明下这两个函数（在 uC/OS-III 中是在 os_cpu.h 中声明），这样才可以在 C 文件里面调用这两个函数。

代码清单 5-27（1）：关中断，NMI 和 HardFault 除外，防止上下文切换被中断。在上下文切换完毕之后，会重新开中断。

代码清单 5-27（2）：将 PSP 的值加载到 R0 寄存器。MRS 是 ARM 32 位数据加载指令，功能是加载特殊功能寄存器的值到通用寄存器。

代码清单 5-27（3）：判断 R0，如果值为 0 则跳转到 OS_CPU_PendSVHandler_nosave。进行第一次任务切换的时候，PSP 在 OSStartHighRdy 初始化为 0，所以这个时候 R0 肯定为 0，则跳转到 OS_CPU_PendSVHandler_nosave。CBZ 是 ARM 16 位转移指令，用于比较，结果为 0 则跳转。

代码清单 5-27（4）：当第一次任务切换的时候，会跳转到这里运行。当执行过一次任务切换之后，则顺序执行到这里。这个标号以后的内容属于下文切换。

代码清单 5-27（5）：加载 OSTCBCurPtr 指针的地址到 R0。在 ARM 汇编中，操作变量都属于间接操作，即要先获取到这个变量的地址。这里 LDR 属于伪指令，不是 ARM 指令。举例：LDR Rd, = label，如果 label 是立即数，那 Rd 等于立即数，如果 label 是一个标识符，比如指针，那存到 Rd 的就是 label 这个标识符的地址。

代码清单 5-27（6）：加载 OSTCBHighRdyPtr 指针的地址到 R1，这里 LDR 也属于伪指令。

代码清单 5-27（7）：加载 OSTCBHighRdyPtr 指针到 R2，这里 LDR 属于 ARM 指令。

代码清单 5-27（8）：存储 OSTCBHighRdyPtr 到 OSTCBCurPtr，实现下一个要运行的任务的 TCB 存储到 OSTCBCurPtr。

代码清单 5-27（9）：加载 OSTCBHighRdyPtr 到 R0。TCB 中第一个成员是栈指针 StkPtr，所以这个时候 R0 等于 StkPtr，后续操作任务栈都是通过操作 R0 来实现，不需要操作 StkPtr。

代码清单 5-27（10）：将任务栈中需要手动加载的内容加载到 CPU 寄存器 R4-R11，同时会递增 R0，让 R0 指向空闲栈的栈顶。LDMIA 中的 I 是 increase 的缩写，A 是 after 的缩小，R0 后面的感叹号“！”表示会自动调节 R0 里面存的指针。当任务被创建的时候，任务的栈会被初始化，初始化的流程是：先让栈指针 StkPtr 指向栈顶，然后从栈顶开始依次存储异常退出时会自动加载到 CPU 寄存器的值和需要手动加载到 CPU 寄存器的值，具体代码实现见代码清单 5-12 OSTaskStkInit()函数，栈空间的分布情况具体见图 5-3。当把需要手动加载到 CPU 的栈内容加载完毕之后，栈空间的分布图和栈指针指向具体见图 5-4，注意这个时候 StkPtr 不变，变的是 R0。

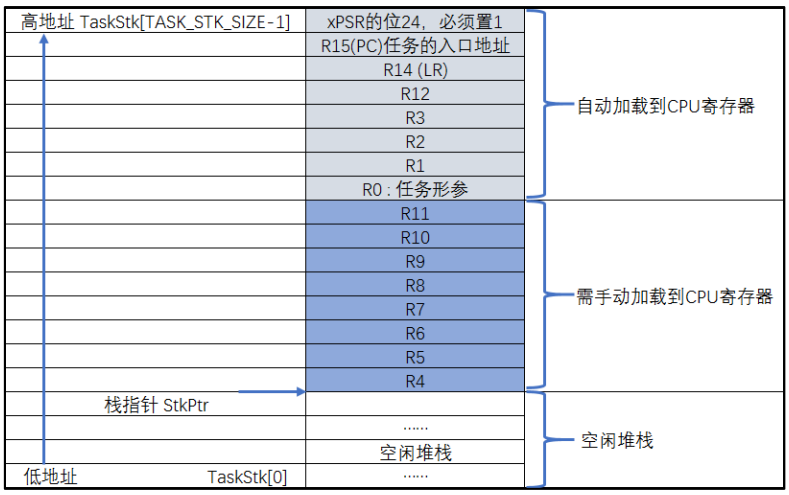


图 5-3 任务创建成功后栈空间的分布图

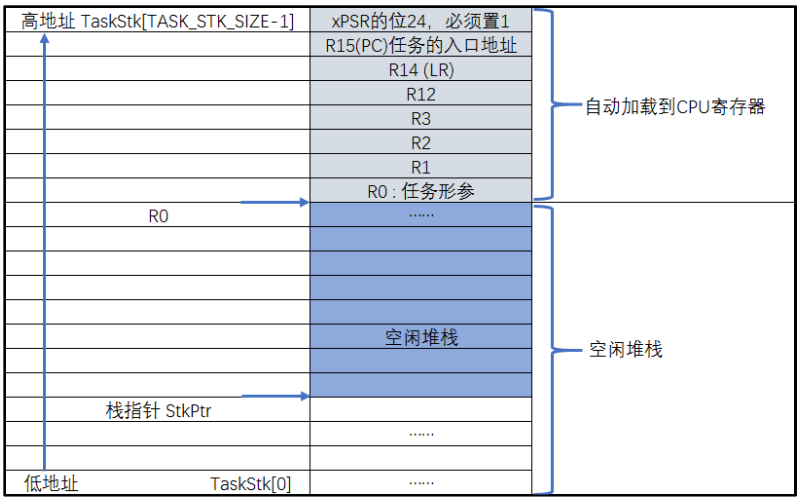


图 5-4 手动加载栈内容到 CPU 寄存器后的栈空间分布图

代码清单 5-27（11）：更新 PSP 的值，这个时候 PSP 与图 5-4 中 R0 的指向一致。

代码清单 5-27（12）：设置 LR 寄存器的位 2 为 1，确保异常退出时使用的堆栈指针是 PSP。当异常退出后，就切换到就绪任务中优先级最高的任务继续运行。

代码清单 5-27（13）：开中断。上下文切换已经完成了四分之三，剩下的就是异常退出时自动保存的部分。

代码清单 5-27（14）：异常返回，这个时候任务堆栈中的剩下内容将会自动加载到 xPSR，PC（任务入口地址），R14，R12，R3，R2，R1，R0（任务的形参）这些寄存器。同时 PSP 的值也将更新，即指向任务堆栈的栈顶。这样就切换到了新的任务。这个时候栈空间的分布具体见图 5-5。

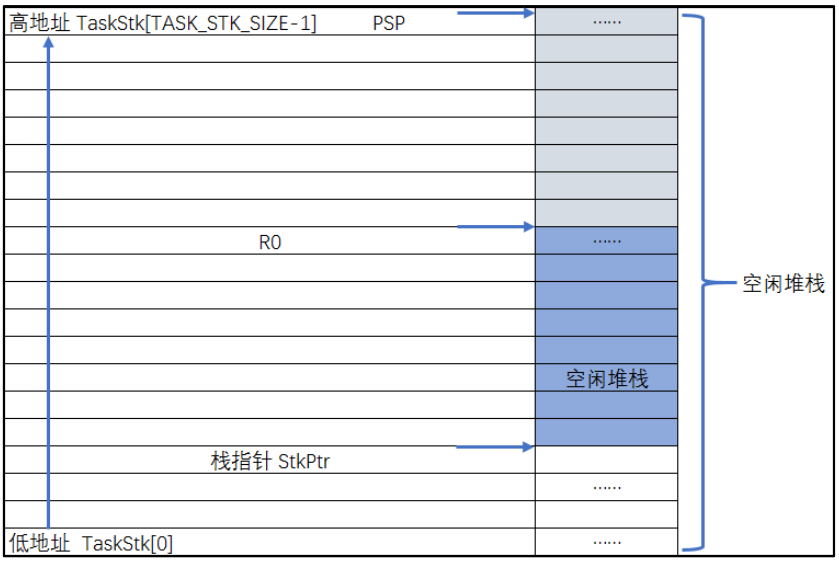


图 5-5 刚切换完成即将运行的任务的栈空间分布和栈指针指向

代码清单 5-27（15）：手动存储 CPU 寄存器 R4-R11 的值到当前任务的堆栈。当异常发生，进入 PendSV 异常服务函数的时候，当前 CPU 寄存器 xPSR，PC（任务入口地址），R14，R12，R3，R2，R1，R0 会自动存储到当前的任务堆栈，同时递减 PSP 的值，这个时候当前任务的堆栈空间分布具体见图 5-6。当执行 STMDB　R0!, {R4-R11}代码后，当前任务的栈空间分布图具体见。

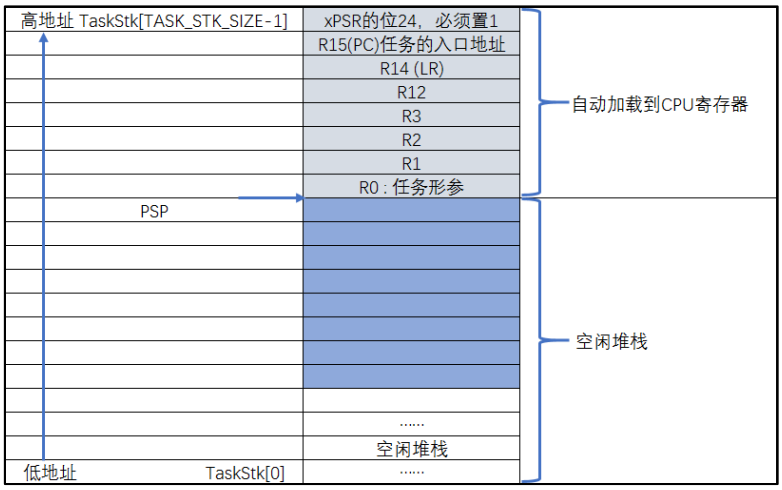


图 5-6 进入 PendSV 异常时，当前任务的栈空间分布

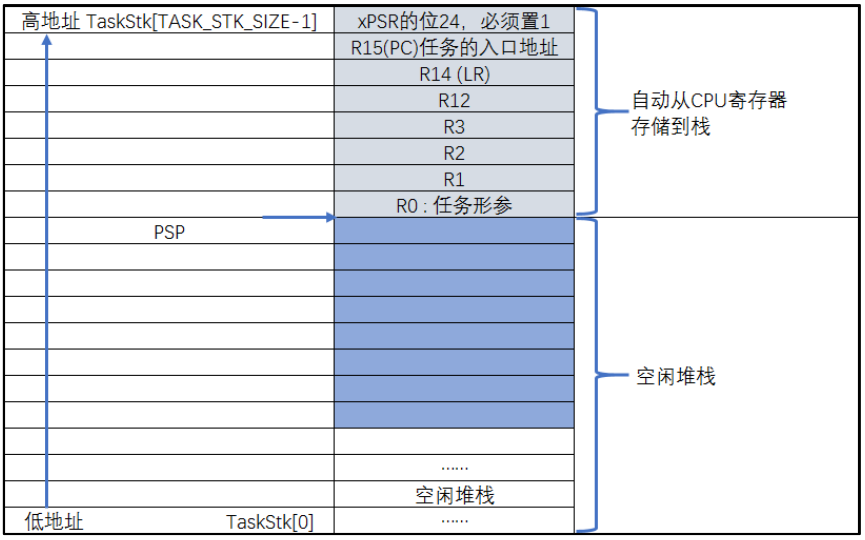


图 5-7 当前任务执行完上文保存时的栈空间分布

代码清单 5-27（16）：加载 OSTCBCurPtr 指针的地址到 R1，这里 LDR 属于伪指令。

代码清单 5-27（17）：加载 OSTCBCurPtr 指针到 R1，这里 LDR 属于 ARM 指令。

代码清单 5-27（18）：存储 R0 的值到 OSTCBCurPtr->OSTCBStkPtr，这个时候 R0 存的是任务空闲栈的栈顶。到了这里，上文的保存就总算完成。这个时候当前任务的堆栈空间分布和栈指针指向具体见图 5-8。

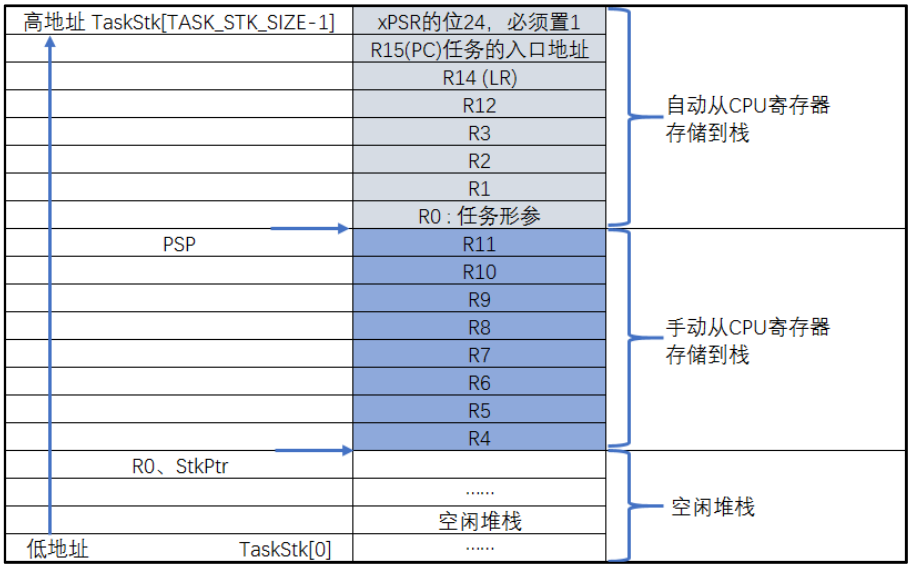


图 5-8 当前任务执行完上文保存时的栈空间分布和 StkPtr 指向

表格 5-3PendSV_Handler 函数中涉及到的 ARM 汇编指令讲解

指令名称	作用
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
CBZ	比较，如果结果为 0 就转移
CBNZ	比较，如果结果非 0 就转移
LDR	从存储器中加载字到一个寄存器中
LDR[伪指令]	加载一个立即数或者一个地址值到一个寄存器。举例：LDR Rd, = label，如果

	label 是立即数，那 Rd 等于立即数，如果 label 是一个标识符，比如指针，那存到 Rd 的就是 label 这个标识符的地址
LDRH	从存储器中加载半字到一个寄存器中
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
STRH	把一个寄存器存器的低半字存储到存储器中
STRB	把一个寄存器的低字节存储到存储器中
LDMIA	加载多个字，并且在加载后自增基址寄存器
STMIA	存储多个字，并且在存储后自增基址寄存器
ORR	按位或
BX	直接跳转到由寄存器给定的地址
BL	跳转到 标号对应的地址，并且把跳转前的下条指令地址保存到 LR
BLX	跳转到由寄存器 REG 给出的的地址，并根据 REG 的 LSB 切换处理器状态，还要把转移前的下条指令地址保存到 LR。ARM(LSB=0)，Thumb(LSB=1)。CM3 只在 Thumb 中运行，就必须保证 reg 的 LSB=1，否则一个 fault 打过来

5.7 main 函数

main 函数在文件 app.c 中编写，其中 app.c 文件中的所有代码具体见代码清单 5-29。

代码清单 5-29 app.c 文件中的代码

```
1  /*
2  *********************************************************************
3  *                               包含的头文件
4  *********************************************************************
5  */
6  #include "os.h"
7  #include "ARMCM3.h"
8
9  /*
10 *********************************************************************
11 *                                 宏定义
12 *********************************************************************
13 */
14
15
16 /*
17 *********************************************************************
18 *                               全局变量
19 *********************************************************************
20 */
21
22 uint32_t flag1;
23 uint32_t flag2;
24
25 /*
26 *********************************************************************
27 *                          TCB & STACK & 任务声明
28 *********************************************************************
29 */
30 #define  TASK1_STK_SIZE        20
31 #define  TASK2_STK_SIZE        20
32
33 static   CPU_STK   Task1Stk[TASK1_STK_SIZE];
34 static   CPU_STK   Task2Stk[TASK2_STK_SIZE];
35
36 static   OS_TCB    Task1TCB;
37 static   OS_TCB    Task2TCB;
```

```
38
39 void     Task1( void *p_arg );
40 void     Task2( void *p_arg );
41
42 /*
43 *********************************************************************
44 *                                函数声明
45 *********************************************************************
46 */
47 void delay(uint32_t count);
48
49 /*
50 *********************************************************************
51 *                                main 函数
52 *********************************************************************
53 */
54 /*
55 * 注意事项：1、该工程使用软件仿真，debug 需选择 Ude Simulator
56 *          2、在 Target 选项卡里面把晶振 Xtal(Mhz)的值改为 25，默认是 12，
57 *               改成 25 是为了跟 system_ARMCM3.c 中定义的__SYSTEM_CLOCK 相同，
58 *               确保仿真的时候时钟一致
59 */
60 int main(void)
61 {
62     OS_ERR err;
63
64
65
66     /* 初始化相关的全局变量 */
67     OSInit(&err);
68
69     /* 创建任务 */
70     OSTaskCreate ((OS_TCB*)       &Task1TCB,
71                   (OS_TASK_PTR ) Task1,
72                   (void *)        0,
73                   (CPU_STK*)      &Task1Stk[0],
74                   (CPU_STK_SIZE) TASK1_STK_SIZE,
75                   (OS_ERR *)      &err);
76
77     OSTaskCreate ((OS_TCB*)       &Task2TCB,
78                   (OS_TASK_PTR ) Task2,
79                   (void *)        0,
80                   (CPU_STK*)      &Task2Stk[0],
81                   (CPU_STK_SIZE) TASK2_STK_SIZE,
82                   (OS_ERR *)      &err);
83
84     /* 将任务加入到就绪列表 */
85     OSRdyList[0].HeadPtr = &Task1TCB;
86     OSRdyList[1].HeadPtr = &Task2TCB;
87
88     /* 启动 OS，将不再返回 */
89     OSStart(&err);
90 }
91
92 /*
93 *********************************************************************
94 *                                函数实现
95 *********************************************************************
96 */
97 /* 软件延时 */
98 void delay (uint32_t count)
99 {
100     for (; count!=0; count--);
101 }
102
```

```
103
104
105 /* 任务 1 */
106 void Task1( void *p_arg )
107 {
108     for ( ;; ) {
109         flag1 = 1;
110         delay( 100 );
111         flag1 = 0;
112         delay( 100 );
113
114         /* 任务切换，这里是手动切换 */
115         OSSched();
116     }
117 }
118
119 /* 任务 2 */
120 void Task2( void *p_arg )
121 {
122     for ( ;; ) {
123         flag2 = 1;
124         delay( 100 );
125         flag2 = 0;
126         delay( 100 );
127
128         /* 任务切换，这里是手动切换 */
129         OSSched();
130     }
131 }
```

代码清单 5-29 中的所有代码在本小节之前都有循序渐进的讲解，这里这是融合在一起放在 main 函数中。其实现在 Task1 和 Task2 并不会真正的自动切换，而是在各自的函数体里面加入了 OSSched()函数来实现手动切换, OSSched()函数的实现具体见代码清单 5-30。

代码清单 5-30OSSched()函数

```
1 /* 任务切换，实际就是触发 PendSV 异常，然后在 PendSV 异常中进行上下文切换 */
2 void OSSched (void)
3 {
4     if ( OSTCBCurPtr == OSRdyList[0].HeadPtr ) {
5         OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
6     } else {
7         OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
8     }
9
10    OS_TASK_SW();
11 }
```

OSSched()函数的调度算法很简单，即如果当前任务是任务 1，那么下一个任务就是任务 2，如果当前任务是任务 2，那么下一个任务就是任务 1，然后再调用 OS_TASK_SW()函数触发 PendSV 异常，然后在 PendSV 异常里面实现任务的切换。在往后的章节中，我们将继续完善，加入 SysTick 中断，从而实现系统调度的自动切换。OS_TASK_SW()函数其实是一个宏定义，具体是往中断及状态控制寄存器 SCB_ICSR 的位 28（PendSV 异常使能位）写入 1，从而触发 PendSV 异常。OS_TASK_SW()函数在 os_cpu.h 文件中实现，os_cpu.h（os_cpu.h 第一次使用需要自行在文件夹 uC-CPU 中新建并添加到工程的 uC-CPU 组）文件内容具体见代码清单 5-31。

代码清单 5-31 os_cpu.h 文件代码清单

```
1 #ifndef  OS_CPU_H
```

```
#define  OS_CPU_H

/*
*********************************************************************
*                                  宏定义
*********************************************************************
*/

#ifndef  NVIC_INT_CTRL
/* 中断控制及状态寄存器 SCB_ICSR */
#define  NVIC_INT_CTRL                  *((CPU_REG32 *)0xE000ED04)
#endif

#ifndef  NVIC_PENDSVSET
/* 触发 PendSV 异常的值 Bit28: PENDSVSET */
#define  NVIC_PENDSVSET                 0x10000000
#endif

/* 触发 PendSV 异常 */
#define  OS_TASK_SW()                   NVIC_INT_CTRL = NVIC_PENDSVSET
/* 触发 PendSV 异常 */
#define  OSIntCtxSw()                   NVIC_INT_CTRL = NVIC_PENDSVSET
/*
*********************************************************************
*                                  函数声明
*********************************************************************
*/
void OSStartHighRdy(void); /* 在 os_cpu_a.s 中实现 */
void PendSV_Handler(void); /* 在 os_cpu_a.s 中实现 */


#endif    /* OS_CPU_H */
```

5.8 实验现象

本章代码讲解完毕，接下来是软件调试仿真，具体过程见图 5-9、图 5-10、图 5-11、图 5-12 和图 5-13。

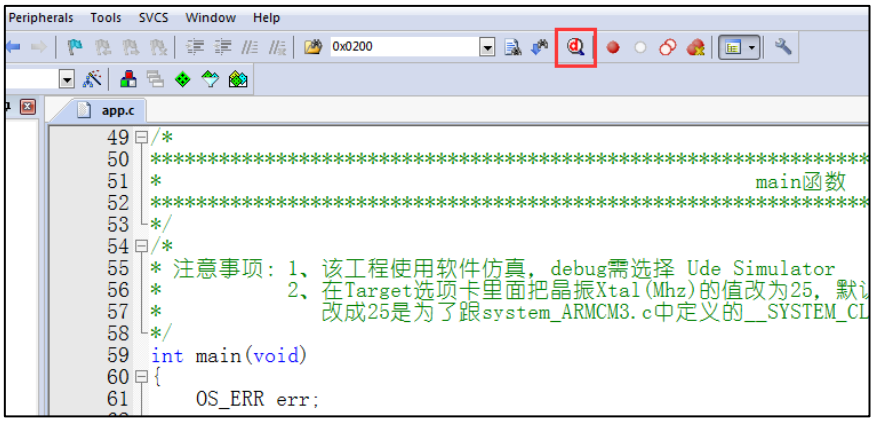


图 5-9 点击 Debug 按钮，进入调试界面

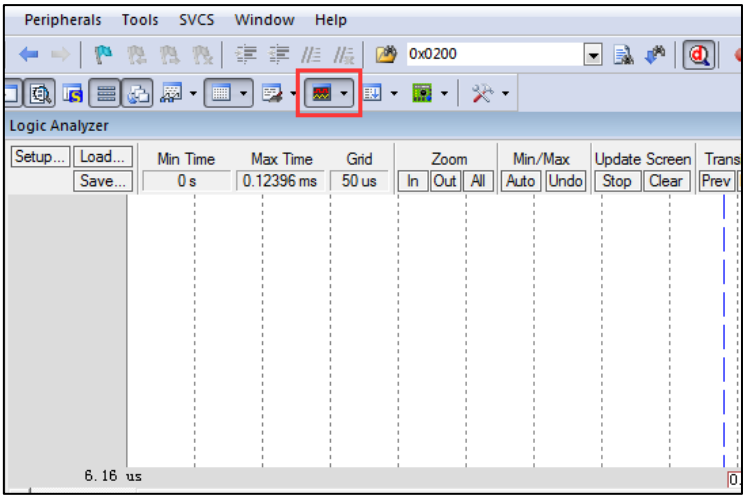


图 5-10 点击逻辑分析仪按钮，调出逻辑分析仪

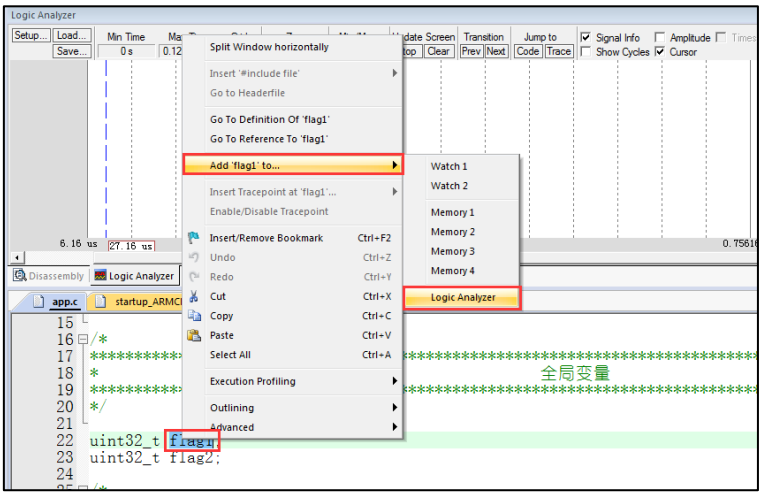


图 5-11 将要观察的变量添加到逻辑分析仪

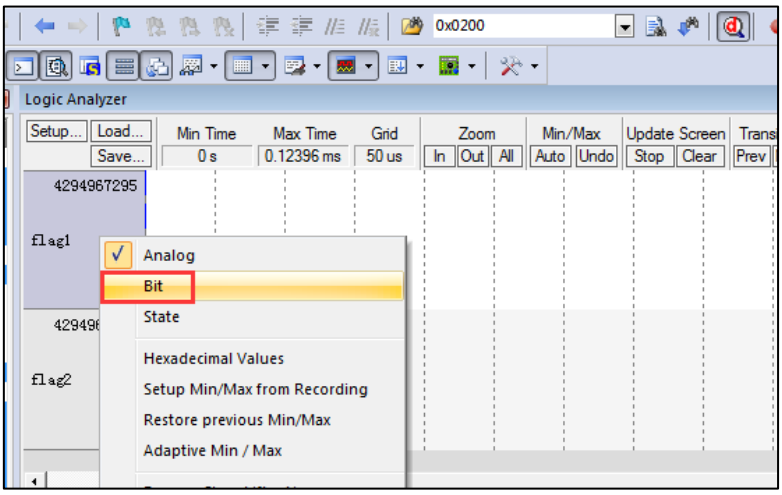


图 5-12 将变量设置为 Bit 模式，默认是 Analog

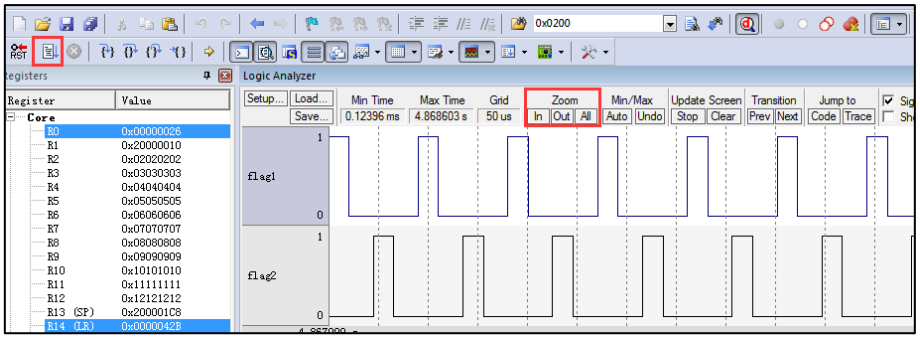


图 5-13 点击全速运行按钮，即可看到波形，Zoom 栏的 In Out All 可放大和缩小波形

至此，本章讲解完毕。但是，只是把本章的内容看完，然后再仿真看看波形是远远不够的，应该是把任务的堆栈、TCB、OSTCBCurPtr 和 OSTCBHighRdyPtr 这些变量统统添加到观察窗口，然后单步执行程序，看看这些变量是怎么变化的。特别是任务切换时，CPU 寄存器、任务堆栈和 PSP 这些是怎么变化的，让机器执行代码的过程在自己的脑子里面过一遍。图 5-14 就是我在仿真调试时的观察窗口。

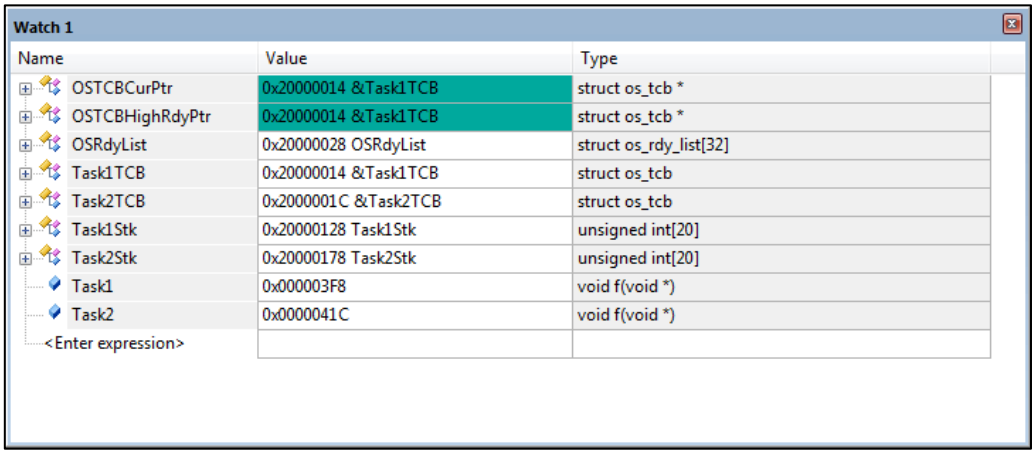


图 5-14 软件调试仿真时的 Watch 窗口

第6章 任务时间片运行

本章在上一章的基础上，加入 SysTick 中断，在 SysTick 中断服务函数里面进行任务切换，从而实现双任务的时间片运行，即每个任务运行的时间都是一样的。

6.1 SysTick 简介

RTOS 需要一个时基来驱动，系统任务调度的频率等于该时基的频率。通常该时基由一个定时器来提供，也可以从其它周期性的信号源获得。刚好 Cortex-M 内核中有一个系统定时器 SysTick，它内嵌在 NVIC 中，是一个 24 位的递减的计数器，计数器每计数一次的时间为 1/SYSCLK。当重装载数值寄存器的值递减到 0 的时候，系统定时器就产生一次中断，以此循环往复。因为 SysTick 是嵌套在内核中的，所以使得 OS 在 Cortex-M 器件中编写的定时器代码不必修改，使移植工作一下子变得简单很多。所以 SysTick 是最适合给操作系统提供时基，用于维护系统心跳的定时器。有关 SysTick 的寄存器汇总见表格 6-1，各个寄存器的用法见表格 6-2、表格 6-3、表格 6-4 和表格 6-5。

表格 6-1SysTick 寄存器汇总

寄存器名称	寄存器描述
CTRL	SysTick 控制及状态寄存器
LOAD	SysTick 重装载数值寄存器
VAL	SysTick 当前数值寄存器
CALIB	SysTick 校准数值寄存器

表格 6-2 SysTick 控制及状态寄存器

位段	名称	类型	复位值	描述
16	COUNTFLAG	R/W	0	如果在上次读取本寄存器后， SysTick 已经计到了 0，则该位为 1。
2	CLKSOURCE	R/W	0	时钟源选择位，0=AHB/8，1=处理器时钟 AHB
1	TICKINT	R/W	0	1=SysTick 倒数计数到 0 时产生 SysTick 异常请求， 0=数到 0 时无动作。也可以通过读取 COUNTFLAG 标志位来确定计数器是否递减到 0
0	ENABLE	R/W	0	SysTick 定时器的使能位

表格 6-3 SysTick 重装载数值寄存器

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数计数至零时，将被重装载的值

表格 6-4 SysTick 当前数值寄存器

位段	名称	类型	复位值	描述
23:0	CURRENT	R/W	0	读取时返回当前倒计数的值，写它则使之清零，同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志

表格 6-5 SysTick 校准数值寄存器

位段	名称	类型	复位值	描述
31	NOREF	R	0	NOREF flag. Reads as zero. Indicates that a separate reference clock is provided. The frequency of this clock is HCLK/8
30	SKEW	R	1	Reads as one. Calibration value for the 1 ms inexact timing is not known because TENMS is not known. This can affect the suitability of SysTick as a software real time clock
23:0	TENMS	R	0	Indicates the calibration value when the SysTick counter runs on HCLK max/8 as external clock. The value is product dependent, please refer to the Product Reference Manual, SysTick Calibration Value section. When HCLK is programmed at the maximum frequency, the SysTick period is 1ms. If calibration information is not known, calculate the calibration value required from the frequency of the processor clock or external clock.

系统定时器的校准数值寄存器在定时实验中不需要用到。有关各个位的描述这里引用手册里面的英文版本，比较晦涩难懂，暂时不知道这个寄存器用来干什么。有研究过的朋友可以交流，起个抛砖引玉的作用。

6.2 初始化 SysTick

使用 SysTick 非常简单，只需一个初始化函数搞定，OS_CPU_SysTickInit 函数在 os_cpu_c.c 中定义，具体实现见代码清单 6-1。在这里，SysTick 初始化函数我们没有使用 uC/OS-III 官方的，我们是自己另外编写了一个。区别是 uC/OS-III 官方的 OS_CPU_SysTickInit 函数里面涉及到 SysTick 寄存器都是重新在 cpu.h 中定义，而我们自己编写的则是使用 ARMCM3.h（记得在 os_cpu_c.c 的开头包含 ARMCM3.h 这个头文件）这个固件库文件里面定义的寄存器，仅此区别而已。

代码清单 6-1 SysTick 初始化

```
1  #if 0 /* 不用 uCOS-III 自带的 */
2  void  OS_CPU_SysTickInit (CPU_INT32U  cnts)
3  {
4      CPU_INT32U  prio;
5
6      /* 填写 SysTick 的重载计数值 */
7      CPU_REG_NVIC_ST_RELOAD = cnts - 1u;
8
9      /* 设置 SysTick 中断优先级 */
10     prio  = CPU_REG_NVIC_SHPRI3;
11     prio &= DEF_BIT_FIELD(24, 0);
12     prio |= DEF_BIT_MASK(OS_CPU_CFG_SYSTICK_PRIO, 24);
13
14     CPU_REG_NVIC_SHPRI3 = prio;
15
16     /* 使能 SysTick 的时钟源和启动计数器 */
17     CPU_REG_NVIC_ST_CTRL |= CPU_REG_NVIC_ST_CTRL_CLKSOURCE |
```

```
18                                  CPU_REG_NVIC_ST_CTRL_ENABLE;
19      /* 使能 SysTick 的定时中断 */
20      CPU_REG_NVIC_ST_CTRL |= CPU_REG_NVIC_ST_CTRL_TICKINT;
21  }
22
23  #else /* 直接使用头文件 ARMCM3.h 里面现有的寄存器定义和函数来实现 */
24  void  OS_CPU_SysTickInit (CPU_INT32U  ms)
25  {
26      /* 设置重装载寄存器的值 */
27      SysTick->LOAD  = ms * SystemCoreClock / 1000 - 1;            (1)
28
29      /* 配置中断优先级为最低 */
30      NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1); (2)
31
32      /* 复位当前计数器的值 */
33      SysTick->VAL   = 0;                                          (3)
34
35      /* 选择时钟源、使能中断、使能计数器 */
36      SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |                (4)
37                       SysTick_CTRL_TICKINT_Msk   |                (5)
38                       SysTick_CTRL_ENABLE_Msk;                    (6)
39  }
40  #endif
```

代码清单 6-1 （1）：配置重装载寄存器的值，我们配合函数形参 ms 来配置，如果需要配置为 10ms 产生一次中断，形参设置为 10 即可。

代码清单 6-1 （2）：配置 SysTick 的优先级，这里配置为 15，即最低。

代码清单 6-1 （3）：复位当前计数器的值。

代码清单 6-1 （4）：选择时钟源，这里选择 SystemCoreClock。

代码清单 6-1 （5）：使能中断。

代码清单 6-1 （6）：使能计数器开始计数。

6.3 编写 SysTick 中断服务函数

SysTick 中断服务函数也是在 os_cpu_c.c 中定义，具体实现见代码清单 6-2。

代码清单 6-2 SysTick 中断服务函数

```
1  /* SysTick 中断服务函数 */
2  void SysTick_Handler(void)
3  {
4      OSTimeTick();
5  }
```

SysTick 中断服务函数很简单，里面仅调用了函数 OSTimeTick()。OSTimeTick()是与时间相关的函数，在 os_time.c（os_time.c 第一次使用需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III Source 组）文件中定义，具体实现见代码清单 6-3。

代码清单 6-3 OSTimeTick()函数

```
1  void  OSTimeTick (void)
2  {
3      /* 任务调度 */
4      OSSched();
5  }
```

OSTimeTick()很简单，里面仅调用了函数 OSSched，OSSched 函数暂时没有修改，与上一章一样，具体见代码清单 6-4。

代码清单 6-4 OSSched 函数

```
1  void OSSched (void)
2  {
3      if ( OSTCBCurPtr == OSRdyList[0].HeadPtr ) {
4          OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
5      } else {
6          OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
7      }
8
9      OS_TASK_SW();
10 }
```

6.4 main 函数

main 函数与上一章区别不大，仅仅是加入了 SysTick 相关的内容，具体见代码清单 6-5。

代码清单 6-5 main 函数和任务代码

```
1  int main(void)
2  {
3      OS_ERR err;
4
5      /* 关闭中断 */
6      CPU_IntDis();                                              (1)
7
8      /* 配置 SysTick 10ms 中断一次 */
9      OS_CPU_SysTickInit (10);                                   (2)
10
11     /* 初始化相关的全局变量 */
12     OSInit(&err);
13
14     /* 创建任务 */
15     OSTaskCreate ((OS_TCB*)       &Task1TCB,
16                   (OS_TASK_PTR ) Task1,
17                   (void *)        0,
18                   (CPU_STK*)      &Task1Stk[0],
19                   (CPU_STK_SIZE) TASK1_STK_SIZE,
20                   (OS_ERR *)      &err);
21
22     OSTaskCreate ((OS_TCB*)       &Task2TCB,
23                   (OS_TASK_PTR ) Task2,
24                   (void *)        0,
25                   (CPU_STK*)      &Task2Stk[0],
26                   (CPU_STK_SIZE) TASK2_STK_SIZE,
27                   (OS_ERR *)      &err);
28
29     /* 将任务加入到就绪列表 */
30     OSRdyList[0].HeadPtr = &Task1TCB;
31     OSRdyList[1].HeadPtr = &Task2TCB;
32
33     /* 启动 OS，将不再返回 */
34     OSStart(&err);
35 }
36
37
38
39 /* 任务 1 */
40 void Task1( void *p_arg )
41 {
42     for ( ;; ) {
43         flag1 = 1;
44         delay( 100 );
45         flag1 = 0;
```

```
46          delay( 100 );
47
48          /* 任务切换，这里是手动切换 */
49          //OSSched();                                    (3)
50      }
51  }
52
53  /* 任务 2 */
54  void Task2( void *p_arg )
55  {
56      for ( ;; ) {
57          flag2 = 1;
58          delay( 100 );
59          flag2 = 0;
60          delay( 100 );
61
62          /* 任务切换，这里是手动切换 */
63          //OSSched();                                    (4)
64      }
65  }
```

代码清单 6-5 （1）：关闭中断。因为在 OS 系统初始化之前我们使能了 SysTick 定时器产生 10ms 的中断，在中断里面触发任务调度，如果一开始我们不关闭中断，就会在 OS 还有启动之前就进入 SysTick 中断，然后发生任务调度，既然 OS 都还没启动，那调度是不允许发生的，所以先关闭中断。系统启动后，中断由 OSStart()函数里面的 OSStartHighRdy()重新开启。

代码清单 6-5 （2）：配置 SysTick 为 10ms 中断一次。任务的调度是在 SysTick 的中断服务函数中完成的，中断的频率越高就意味着 OS 的调度越高，系统的负荷就越重，一直在不断的进入中断，则执行任务的时间就减小。选择合适的 SysTick 中断频率会提供系统的运行效率，uC/OS-III 官方推荐为 10ms，或者高点也行。

代码清单 6-5 （3）、（4）：任务调度将不再在各自的任务里面实现，而是放到了 SysTick 中断服务函数中。从而实现每个任务都运行相同的时间片，平等的享有 CPU。

6.5 实验想像

进入软件调试，点击全速运行按钮就可看到实验波形，具体见图 6-1。

图 6-1 实验现象

从图 6-1 我们可以看到，两个任务轮流的占有 CPU，享有相同的时间片。其实目前的实验现象与上一章的实验现象还没有本质上的区别，加入 SysTick 只是为了后续章节做准备。上一章两个任务也是轮流的占有 CPU，也是享有相同的时间片，该时间片是任务单次运行的时间。不同的是本章任务的时间片等于 SysTick 定时器的时基，是很多个 任务单次运行时间的综合。即在这个时间片里面任务运行了非常多次，如果我们把波形放大，就会发现大波形里面包含了很多小波形，具体见图 6-2。

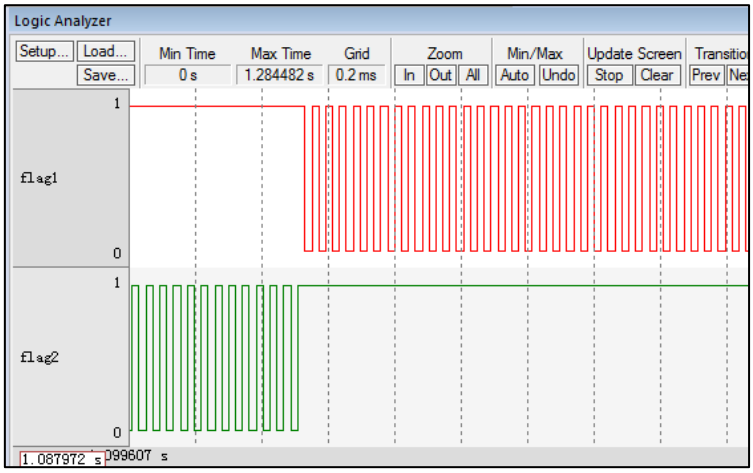


图 6-2 实验现象 2

第7章　阻塞延时与空闲任务

在上一章节中，任务体内的延时使用的是软件延时，即还是让 CPU 空等来达到延时的效果。使用 RTOS 的很大优势就是榨干 CPU 的性能，永远不能让它闲着，任务如果需要延时也就不能再让 CPU 空等来实现延时的效果。RTOS 中的延时叫阻塞延时，即任务需要延时的时候，任务会放弃 CPU 的使用权，CPU 可以去干其它的事情，当任务延时时间到，重新获取 CPU 使用权，任务继续运行，这样就充分地利用了 CPU 的资源，而不是干等着。

当任务需要延时，进入阻塞状态，那 CPU 又去干什么事情了？如果没有其它任务可以运行，RTOS 都会为 CPU 创建一个空闲任务，这个时候 CPU 就运行空闲任务。在 uC/OS-III 中，空闲任务是系统在初始化的时候创建的优先级最低的任务，空闲任务主体很简单，只是对一个全局变量进行计数。鉴于空闲任务的这种特性，在实际应用中，当系统进入空闲任务的时候，可在空闲任务中让单片机进入休眠或者低功耗等操作。

7.1 实现空闲任务

7.1.1 定义空闲任务堆栈

空闲任务堆栈在 os_cfg_app.c（os_cfg_app.c 第一次使用需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III Source 组）文件中定义，具体见代码清单 7-1。

代码清单 7-1 os_cfg_app.c 文件代码

```
1  /*
2  ************************************************************************
3  *                                  数据域
4  ************************************************************************
5  */
6
7  CPU_STK     OSCfg_IdleTaskStk[OS_CFG_IDLE_TASK_STK_SIZE];          (1)
8
9
10
11 /*
12 ************************************************************************
13 *                                  常量
14 ************************************************************************
15 */
16
17 /* 空闲任务堆栈起始地址 */
18 CPU_STK       * const  OSCfg_IdleTaskStkBasePtr    = \             (2)
19         (CPU_STK     *)&OSCfg_IdleTaskStk[0];
20 /* 空闲任务堆栈大小 */
21 CPU_STK_SIZE    const  OSCfg_IdleTaskStkSize       = \
22         (CPU_STK_SIZE)OS_CFG_IDLE_TASK_STK_SIZE;
```

代码清单 7-1 （1）：空闲任务的堆栈是一个定义好的数组，大小由 OS_CFG_IDLE_TASK_STK_SIZE 这个宏控制。OS_CFG_IDLE_TASK_STK_SIZE 在 os_cfg_app.h 这个头文件定义，大小为 128，具体见代码清单 7-2。

代码清单 7-2 os_cfg_app.h 文件代码

```
#ifndef OS_CFG_APP_H
#define OS_CFG_APP_H

/*
************************************************************************
*                                  常量
************************************************************************
*/

/* 空闲任务堆栈大小 */
#define  OS_CFG_IDLE_TASK_STK_SIZE        128u

#endif  /* OS_CFG_APP_H */
```

代码清单 7-1 （2）：空闲任务的堆栈的起始地址和大小均被定义成一个常量，不能被修改。变量 OSCfg_IdleTaskStkBasePtr 和 OSCfg_IdleTaskStkSize 同时还在 os.h 中声明，这样就具有全局属性，可以在其它文件里面被使用，具体声明见代码清单 7-3。

代码清单 7-3 OSCfg_IdleTaskStkBasePtr 和 OSCfg_IdleTaskStkSize 声明

```
/* 空闲任务堆栈起始地址 */
extern CPU_STK       * const  OSCfg_IdleTaskStkBasePtr;
/* 空闲任务堆栈大小 */
extern CPU_STK_SIZE    const  OSCfg_IdleTaskStkSize;
```

7.1.2 定义空闲任务 TCB

任务控制块 TCB 是每一个任务必须的，空闲任务的 TCB 在 os.h 中定义，是一个全局变量，具体见代码清单 7-4。

代码清单 7-4 定义空闲任务 TCB

```
/* 空闲任务 TCB */
OS_EXT     OS_TCB          OSIdleTaskTCB;
```

7.1.3 定义空闲任务函数

空闲任务正如其名，空闲，任务体里面只是对全局变量 OSIdleTaskCtr ++ 操作，具体实现见代码清单 7-5。

代码清单 7-5 空闲任务函数

```
/* 空闲任务 */
void  OS_IdleTask (void  *p_arg)
{
    p_arg = p_arg;

    /* 空闲任务什么都不做，只对全局变量 OSIdleTaskCtr ++ 操作 */
    for (;;) {
        OSIdleTaskCtr++;
    }
```

```
10 }
```

代码清单 7-5 中的全局变量 OSIdleTaskCtr 在 os.h 中定义，具体见代码清单 7-6。

代码清单 7-6 OSIdleTaskCtr 定义

```
/* 空闲任务计数变量 */
1 OS_EXT     OS_IDLE_CTR     OSIdleTaskCtr;
```

代码清单 7-6 中的 OS_IDLE_CTR 是在 os_type.h 中重新定义的数据类型，具体见代码清单 7-7。

代码清单 7-7 OS_IDLE_CTR 定义

```
/* 空闲任务计数变量定义 */
1 typedef    CPU_INT32U       OS_IDLE_CTR;
```

7.1.4 空闲任务初始化

空闲任务的初始化在 OSInit()在完成，意味着在系统还没有启动之前空闲任务就已经创建好，具体在 os_core.c 定义，具体代码见代码清单 7-8。

代码清单 7-8 空闲任务初始化函数

```
 1 void OSInit (OS_ERR *p_err)
 2 {
 3     /* 配置 OS 初始状态为停止态 */
 4     OSRunning =  OS_STATE_OS_STOPPED;
 5
 6     /* 初始化两个全局 TCB，这两个 TCB 用于任务切换 */
 7     OSTCBCurPtr = (OS_TCB *)0;
 8     OSTCBHighRdyPtr = (OS_TCB *)0;
 9
10     /* 初始化就绪列表 */
11     OS_RdyListInit();
12
13     /* 初始化空闲任务 */
14     OS_IdleTaskInit(p_err);                                          (1)
15     if (*p_err != OS_ERR_NONE) {
16         return;
17     }
18 }
19
20 /* 空闲任务初始化 */
21 void  OS_IdleTaskInit(OS_ERR  *p_err)
22 {
23     /* 初始化空闲任务计数器 */
24     OSIdleTaskCtr = (OS_IDLE_CTR)0;                                  (2)
25
26     /* 创建空闲任务 */
27     OSTaskCreate( (OS_TCB     *)&OSIdleTaskTCB,                      (3)
28                   (OS_TASK_PTR )OS_IdleTask,
29                   (void       *)0,
30                   (CPU_STK    *)OSCfg_IdleTaskStkBasePtr,
31                   (CPU_STK_SIZE)OSCfg_IdleTaskStkSize,
32                   (OS_ERR     *)p_err );
33 }
```

代码清单 7-8（1）：空闲任务初始化函数在 OSInit 中调用，在系统还没有启动之前就被创建。

代码清单 7-8（2）：初始化空闲任务计数器，我们知道，这个是预先在 os.h 中定义好的全局变量。

代码清单 7-8（3）：创建空闲任务，把堆栈，TCB，任务函数联系在一起。

7.2 实现阻塞延时

阻塞延时的阻塞是指任务调用该延时函数后，任务会被剥离 CPU 使用权，然后进入阻塞状态，直到延时结束，任务重新获取 CPU 使用权才可以继续运行。在任务阻塞的这段时间，CPU 可以去执行其它的任务，如果其它的任务也在延时状态，那么 CPU 就将运行空闲任务。阻塞延时函数在 os_time.c 中定义，具体代码实现见代码清单 7-9。

代码清单 7-9 阻塞延时代码

```
1 /* 阻塞延时 */
2 void  OSTimeDly(OS_TICK dly)
3 {
4     /* 设置延时时间 */
5     OSTCBCurPtr->TaskDelayTicks = dly;                    (1)
6 
7     /* 进行任务调度 */
8     OSSched();                                            (2)
9 }
```

代码清单 7-9 （1）：TaskDelayTicks 是任务控制块的一个成员，用于记录任务需要延时的时间，单位为 SysTick 的中断周期。比如我们本书当中 SysTick 的中断周期为 10ms，调用 OSTimeDly(2)则完成 2*10ms 的延时。TaskDelayTicks 的定义具体见代码清单 7-10。

代码清单 7-10 TaskDelayTicks 定义

```
1 struct os_tcb {
2     CPU_STK         *StkPtr;
3     CPU_STK_SIZE    StkSize;
4 
5     /* 任务延时周期个数 */
6     OS_TICK         TaskDelayTicks;
7 };
```

代码清单 7-9 （2）：任务调度。这个时候的任务调度与上一章节的不一样，具体见代码清单 7-11，其中加粗部分为上一章节的代码，现已用条件编译屏蔽掉。

代码清单 7-11 任务调度

```
1 void OSSched(void)
2 {
3 #if 0 /* 非常简单的任务调度：两个任务轮流执行 */
4     if ( OSTCBCurPtr == OSRdyList[0].HeadPtr ) {
5         OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
6     } else {
7         OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
8     }
9 #endif
10 
```

```
11      /* 如果当前任务是空闲任务，那么就去尝试执行任务 1 或者任务 2，
12         看看他们的延时时间是否结束，如果任务的延时时间均没有到期，
13         那就返回继续执行空闲任务 */
14      if ( OSTCBCurPtr == &OSIdleTaskTCB ) {                              (1)
15          if (OSRdyList[0].HeadPtr->TaskDelayTicks == 0) {
16              OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
17          } else if (OSRdyList[1].HeadPtr->TaskDelayTicks == 0) {
18              OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
19          } else {
20              /* 任务延时均没有到期则返回，继续执行空闲任务 */
21              return;
22          }
23      } else {                                                            (2)
24          /*如果是 task1 或者 task2 的话，检查下另外一个任务，
25            如果另外的任务不在延时中，就切换到该任务
26           否则，判断下当前任务是否应该进入延时状态，
27           如果是的话，就切换到空闲任务。否则就不进行任何切换 */
28          if (OSTCBCurPtr == OSRdyList[0].HeadPtr) {
29              if (OSRdyList[1].HeadPtr->TaskDelayTicks == 0) {
30                  OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
31              } else if (OSTCBCurPtr->TaskDelayTicks != 0) {
32                  OSTCBHighRdyPtr = &OSIdleTaskTCB;
33              } else {
34                  /* 返回，不进行切换，因为两个任务都处于延时中 */
35                  return;
36              }
37          } else if (OSTCBCurPtr == OSRdyList[1].HeadPtr) {
38              if (OSRdyList[0].HeadPtr->TaskDelayTicks == 0) {
39                  OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
40              } else if (OSTCBCurPtr->TaskDelayTicks != 0) {
41                  OSTCBHighRdyPtr = &OSIdleTaskTCB;
42              } else {
43                  /* 返回，不进行切换，因为两个任务都处于延时中 */
44                  return;
45              }
46          }
47      }
48
49      /* 任务切换 */
50      OS_TASK_SW();                                                       (3)
51  }
```

代码清单 7-11（1）：如果当前任务是空闲任务，那么就去尝试执行任务 1 或者任务 2，看看他们的延时时间是否结束，如果任务的延时时间均没有到期，那就返回继续执行空闲任务。

代码清单 7-11（2）：如果当前任务不是空闲任务则会执行到此，那就看看当前任务是哪个任务。无论是哪个任务，都要检查下另外一个任务是否在延时中，如果没有在延时，那就切换到该任务，如果有在延时，那就判断下当前任务是否应该进入延时状态，如果是的话，就切换到空闲任务。否则就不进行任务切换 。

代码清单 7-11（3）：任务切换，实际就是触发 PendSV 异常。

7.3 main 函数

main 函数和任务代码变动不大，具体见代码清单 7-12，有变动部分代码已加粗。

代码清单 7-12 main 函数

```
1  int main(void)
```

```
2  {
3      OS_ERR err;
4
5      /* 关闭中断 */
6      CPU_IntDis();
7
8      /* 配置 SysTick 10ms 中断一次 */
9      OS_CPU_SysTickInit (10);
10
11     /* 初始化相关的全局变量 */
12     OSInit(&err);                                                    (1)
13
14     /* 创建任务 */
15     OSTaskCreate ((OS_TCB*)       &Task1TCB,
16                   (OS_TASK_PTR ) Task1,
17                   (void *)       0,
18                   (CPU_STK*)     &Task1Stk[0],
19                   (CPU_STK_SIZE) TASK1_STK_SIZE,
20                   (OS_ERR *)     &err);
21
22     OSTaskCreate ((OS_TCB*)       &Task2TCB,
23                   (OS_TASK_PTR ) Task2,
24                   (void *)       0,
25                   (CPU_STK*)     &Task2Stk[0],
26                   (CPU_STK_SIZE) TASK2_STK_SIZE,
27                   (OS_ERR *)     &err);
28
29     /* 将任务加入到就绪列表 */
30     OSRdyList[0].HeadPtr = &Task1TCB;
31     OSRdyList[1].HeadPtr = &Task2TCB;
32
33     /* 启动 OS，将不再返回 */
34     OSStart(&err);
35 }
36
37 /* 任务 1 */
38 void Task1( void *p_arg )
39 {
40     for ( ;; ) {
41         flag1 = 1;
42         //delay( 100 );
43         OSTimeDly(2);                                                (2)
44         flag1 = 0;
45         //delay( 100 );
46         OSTimeDly(2);
47
48         /* 任务切换，这里是手动切换 */
49         //OSSched();
50     }
51 }
52
53 /* 任务 2 */
54 void Task2( void *p_arg )
55 {
56     for ( ;; ) {
57         flag2 = 1;
58         //delay( 100 );
59         OSTimeDly(2);                                                (3)
60         flag2 = 0;
61         //delay( 100 );
62         OSTimeDly(2);
63
64         /* 任务切换，这里是手动切换 */
65         //OSSched();
66     }
```

```
67 }
```

代码清单 7-12（1）：空闲任务初始化函数在 OSInint 中调用，在系统启动之前创建好空闲任务。

代码清单 7-12（2）和（3）：延时函数均替代为阻塞延时，延时时间均为 2 个 SysTick 中断周期，即 20ms。

7.4 实验现象

进入软件调试，全速运行程序，从逻辑分析仪中可以看到两个任务的波形是完全同步，就好像 CPU 在同时干两件事情，具体仿真的波形图见图 7-1 和图 7-2。

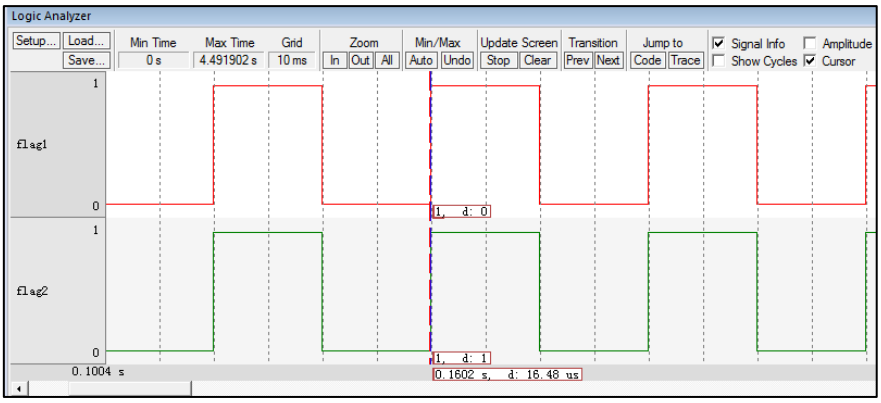


图 7-1 实验现象 1

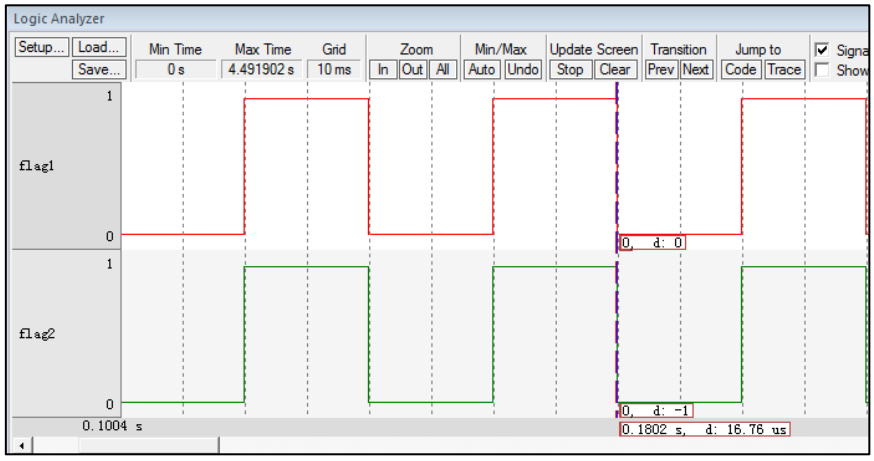


图 7-2 实验现象 2

从图 7-1 和图 7-2 可以看出，flag1 和 flag2 的高电平的时间为(0.1802-0.1602)s，刚好等于阻塞延时的 20ms，所以实验现象跟代码要实现的功能是一致的。

第8章　时间戳

本章实现时间戳用的是 ARM Cortex-M 系列内核中的 DWT 这个外设的功能，有关这个外设的功能和寄存器说明具体见手册“STM32F10xxx Cortex-M3 programming manual”

8.1 时间戳简介

在 uC/OS-III 中，很多地方的代码都加入了时间测量的功能，比如任务关中断的时间，关调度器的时间等。知道了某段代码的运行时间，就明显地知道该代码的执行效率，如果时间过长就可以优化或者调整代码策略。如果要测量一段代码 A 的时间，那么可以在代码段 A 运行前记录一个时间点 TimeStart，在代码段 A 运行完记录一个时间点 TimeEnd，那么代码段 A 的运行时间 TimeUse 就等于 TimeEnd 减去 TimeStart。这里面的两个时间点 TimeEnd 和 TimeStart，就叫做时间戳，时间戳实际上就是一个时间点。

8.2 时间戳的实现

通常执行一条代码是需要多个时钟周期的，即是 ns 级别。要想准确测量代码的运行时间，时间戳的精度就很重要。通常单片机中的硬件定时器的精度都是 us 级别，远达不到测量几条代码运行时间的精度。

在 ARM Cortex-M 系列内核中，有一个 DWT 的外设，该外设有一个 32 位的寄存器叫 CYCCNT，它是一个向上的 计数器，记录的是内核时钟 HCLK 运行的个数，当 CYCCNT 溢出之后，会清 0 重新开始向上计数。该计数器在 uC/OS-III 中正好被用来实现时间戳的功能。

在 STM32F103 系列的单片机中，HCLK 时钟最高为 72M，单个时钟的周期为 1/72 us = 0.0139us = 14ns，CYCCNT 总共能记录的时间为 2^{32}*14=60S。在 uC/OS-III 中，要测量的时间都是很短的，都是 ms 级别，根本不需要考虑计时器溢出的问题。如果内核代码执行的时间超过 s 的级别，那就背离了实时操作系统实时的设计初衷了，没有意义。

8.3 时间戳代码讲解

8.3.1 CPU_Init()函数

CPU_Init()函数在 cpu_core.c（cpu_core.c 文件第一次使用需要自行在文件夹 uC-CPU 中新建并添加到工程的 uC-CPU 组）中实现，主要做三件事：1、初始化时间戳，2、初始化中断失能时间测量，3、初始化 CPU 名字。第 2 和 3 个功能目前还没有使用到，只实现了第 1 个初始化时间戳的代码，具体见代码清单 8-1。

代码清单 8-1 CPU_Init()函数

```
1 /* CPU 初始化函数 */
2 void  CPU_Init (void)
3 {
4     /* CPU 初始化函数中总共做了三件事
```

```
5          1、初始化时间戳
6          2、初始化中断失能时间测量
7          3、初始化 CPU 名字
8          这里只讲时间戳功能，剩下两个的初始化代码则删除不讲 */
9
10 #if ((CPU_CFG_TS_EN     == DEF_ENABLED) || \                          (1)
11      (CPU_CFG_TS_TMR_EN == DEF_ENABLED))
12     CPU_TS_Init();                                                    (2)
13 #endif
14
15 }
```

代码清单 8-1（1）：CPU_CFG_TS_EN 和 CPU_CFG_TS_TMR_EN 这两个宏在 cpu_core.h 中定义，用于控制时间戳相关的功能代码，具体定义见代码清单 8-2。

代码清单 8-2 CPU_CFG_TS_EN 和 CPU_CFG_TS_TMR_EN 宏定义

```
1 #if    ((CPU_CFG_TS_32_EN == DEF_ENABLED) || \                         (1)
2          (CPU_CFG_TS_64_EN == DEF_ENABLED))
3 #define  CPU_CFG_TS_EN                          DEF_ENABLED
4 #else
5 #define  CPU_CFG_TS_EN                          DEF_DISABLED
6 #endif
7
8 #if    ((CPU_CFG_TS_EN == DEF_ENABLED) || \
9 (defined(CPU_CFG_INT_DIS_MEAS_EN)))
10 #define  CPU_CFG_TS_TMR_EN                      DEF_ENABLED
11 #else
12 #define  CPU_CFG_TS_TMR_EN                      DEF_DISABLED
13 #endif
```

代码清单 8-2（1）：CPU_CFG_TS_32_EN 和 CPU_CFG_TS_64_EN 这两个宏在 cpu_cfg.h（cpu_cfg.h 文件第一次使用需要自行在文件夹 uC-CPU 中新建并添加到工程的 uC-CPU 组）文件中定义，用于控制时间戳是 32 位还是 64 位的，默认使能 32 位，具体见代码清单 8-3。

代码清单 8-3 CPU_CFG_TS_32_EN 和 CPU_CFG_TS_64_EN 宏定义

```
1 #ifndef  CPU_CFG_MODULE_PRESENT
2 #define  CPU_CFG_MODULE_PRESENT
3
4
5 #define  CPU_CFG_TS_32_EN                       DEF_ENABLED
6 #define  CPU_CFG_TS_64_EN                       DEF_DISABLED
7
8 #define  CPU_CFG_TS_TMR_SIZE                    CPU_WORD_SIZE_32
9
10
11 #endif /* CPU_CFG_MODULE_PRESENT */
```

8.3.2 CPU_TS_Init()函数

代码清单 8-1（2）：CPU_TS_Init()是时间戳初始化函数，在 cpu_core.c 中实现，具体见代码清单 8-4。

代码清单 8-4 CPU_TS_Init()函数

```
1 #if ((CPU_CFG_TS_EN     == DEF_ENABLED) || \
2      (CPU_CFG_TS_TMR_EN == DEF_ENABLED))
3 static  void  CPU_TS_Init (void)
4 {
5
6 #if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
7     CPU_TS_TmrFreq_Hz   = 0u;                          (1)
```

```
    CPU_TS_TmrInit();                                   (2)
#endif

}
#endif
```

代码清单 8-4 （1）：CPU_TS_TmrFreq_Hz 是一个在 cpu_core.h 中定义的全局变量，表示 CPU 的系统时钟，具体大小跟硬件相关，如果使用 STM32F103 系列，那就等于 72000000HZ。CPU_TS_TmrFreq_Hz 变量的定义和时间戳相关的数据类型的定义具体见代码清单 8-5。

代码清单 8-5 CPU_TS_TmrFreq_Hz 和时间戳相关的数据类型定义

```
/*
*********************************************************************
*                                  EXTERNS
*                            在 cpu_core.h 开头定义
*********************************************************************
*/

#ifdef   CPU_CORE_MODULE  /* CPU_CORE_MODULE 只在 cpu_core.c 文件的开头定义 */
#define  CPU_CORE_EXT
#else
#define  CPU_CORE_EXT  extern
#endif

/*
*********************************************************************
*                              时间戳数据类型
*                            在 cpu_core.h 文件定义
*********************************************************************
*/

typedef  CPU_INT32U  CPU_TS32;

typedef  CPU_INT32U  CPU_TS_TMR_FREQ;
typedef  CPU_TS32    CPU_TS;
typedef  CPU_INT32U  CPU_TS_TMR;


/*
*********************************************************************
*                              全局变量
*                         在 cpu_core.h 文件定义
*********************************************************************
*/

#if  (CPU_CFG_TS_TMR_EN    == DEF_ENABLED)
CPU_CORE_EXT  CPU_TS_TMR_FREQ  CPU_TS_TmrFreq_Hz;
#endif
```

8.3.3 CPU_TS_TmrInit()函数

代码清单 8-4（2）：时间戳定时器初始化函数 CPU_TS_TmrInit()在 cpu_core.c 实现，具体见代码清单 8-6。

代码清单 8-6 CPU_TS_TmrInit()函数

```
/* 时间戳定时器初始化 */
#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
void  CPU_TS_TmrInit (void)
{
    CPU_INT32U  fclk_freq;
```

```
6
7
8       fclk_freq = BSP_CPU_ClkFreq();                                  (2)
9
10      /* 使能 DWT 外设 */
11      BSP_REG_DEM_CR      |= (CPU_INT32U)BSP_BIT_DEM_CR_TRCENA;       (1)
12      /* DWT CYCCNT 寄存器计数清 0 */
13      BSP_REG_DWT_CYCCNT  = (CPU_INT32U)0u;
14      /* 注意：当使用软件仿真全速运行的时候，会先停在这里，
15                就好像在这里设置了一个断点一样，需要手动运行才能跳过，
16                当使用硬件仿真的时候却不会 */
17      /* 使能 Cortex-M3 DWT CYCCNT 寄存器 */
18      BSP_REG_DWT_CR      |= (CPU_INT32U)BSP_BIT_DWT_CR_CYCCNTENA;
19
20      CPU_TS_TmrFreqSet((CPU_TS_TMR_FREQ)fclk_freq);                  (3)
21  }
22  #endif
```

代码清单 8-6（1）：初始化时间戳计数器 CYCCNT，使能 CYCCNT 计数的操作步骤：

1、先使能 DWT 外设，这个由另外内核调试寄存器 DEMCR 的位 24 控制，写 1 使能。

2、使能 CYCCNT 寄存器之前，先清 0。

3、使能 CYCCNT 寄存器，这个由 DWT_CTRL(代码上宏定义为 DWT_CR)的位 0 控制，写 1 使能。这三个步骤里面涉及到的寄存器定义在 cpu_core.c 文件的开头，具体见代码清单 8-7。

代码清单 8-7 DWT 外设相关寄存器定义

```
1  /*
2  *********************************************************************
3  *                                寄存器定义
4  *********************************************************************
5  */
6  #define  BSP_REG_DEM_CR                    (*(CPU_REG32 *)0xE000EDFC)
7  #define  BSP_REG_DWT_CR                    (*(CPU_REG32 *)0xE0001000)
8  #define  BSP_REG_DWT_CYCCNT                (*(CPU_REG32 *)0xE0001004)
9  #define  BSP_REG_DBGMCU_CR                 (*(CPU_REG32 *)0xE0042004)
10
11 /*
12 *********************************************************************
13 *                                寄存器位定义
14 *********************************************************************
15 */
16
17 #define  BSP_DBGMCU_CR_TRACE_IOEN_MASK                  0x10
18 #define  BSP_DBGMCU_CR_TRACE_MODE_ASYNC                 0x00
19 #define  BSP_DBGMCU_CR_TRACE_MODE_SYNC_01               0x40
20 #define  BSP_DBGMCU_CR_TRACE_MODE_SYNC_02               0x80
21 #define  BSP_DBGMCU_CR_TRACE_MODE_SYNC_04               0xC0
22 #define  BSP_DBGMCU_CR_TRACE_MODE_MASK                  0xC0
23
24 #define  BSP_BIT_DEM_CR_TRCENA                         (1<<24)
25
26 #define  BSP_BIT_DWT_CR_CYCCNTENA                      (1<<0)
```

8.3.4 BSP_CPU_ClkFreq()函数

代码清单 8-6（2）：BSP_CPU_ClkFreq()是一个用于获取 CPU 的 HCLK 时钟的 BSP 函数，具体跟硬件相关，目前只是使用软件仿真，则把硬件相关的代码注释掉，直接手动设

置 CPU 的 HCLK 的时钟等于软件仿真的时钟 25000000HZ。BSP_CPU_ClkFreq()在 cpu_core.c 实现，具体定义见代码清单 8-8。

代码清单 8-8 BSP_CPU_ClkFreq()函数

```
/* 获取 CPU 的 HCLK 时钟
这个是跟硬件相关的，目前我们是软件仿真，我们暂时把跟硬件相关的代码屏蔽掉，
直接手动设置 CPU 的 HCLK 时钟*/
CPU_INT32U  BSP_CPU_ClkFreq (void)
{
#if 0
    RCC_ClocksTypeDef  rcc_clocks;


    RCC_GetClocksFreq(&rcc_clocks);
    return ((CPU_INT32U)rcc_clocks.HCLK_Frequency);
#else
    CPU_INT32U    CPU_HCLK;


    /* 目前软件仿真我们使用 25M 的系统时钟 */
    CPU_HCLK = 25000000;

    return CPU_HCLK;
#endif
}
```

8.3.5 CPU_TS_TmrFreqSet()函数

代码清单 8-6（3）：CPU_TS_TmrFreqSet()函数在 cpu_core.c 定义，具体的作用是把函数 BSP_CPU_ClkFreq()获取到的 CPU 的 HCLK 时钟赋值给全局变量 CPU_TS_TmrFreq_Hz，具体实现见代码清单 8-9。

代码清单 8-9 CPU_TS_TmrFreqSet()函数

```
/* 初始化 CPU_TS_TmrFreq_Hz，这个就是系统的时钟，单位为 HZ */
#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
void  CPU_TS_TmrFreqSet (CPU_TS_TMR_FREQ  freq_hz)
{
    CPU_TS_TmrFreq_Hz = freq_hz;
}
#endif
```

8.3.6 CPU_TS_TmrRd()函数

CPU_TS_TmrRd()函数用于获取 CYCNNT 计数器的值，在 cpu_core.c 中定义，具体实现见代码清单 8-10。

代码清单 8-10 CPU_TS_TmrRd()函数

```
#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
CPU_TS_TMR  CPU_TS_TmrRd (void)
{
    CPU_TS_TMR  ts_tmr_cnts;


    ts_tmr_cnts = (CPU_TS_TMR)BSP_REG_DWT_CYCCNT;

    return (ts_tmr_cnts);
}
```

```
11 #endif
```

8.3.7 OS_TS_GET()函数

OS_TS_GET()函数用于获取 CYCNNT 计数器的值，实际上是一个宏定义，将 CPU 底层的函数 CPU_TS_TmrRd()重新取个名字封装，供内核和用户函数使用，在 os_cpu.h 头文件定义，具体实现见代码清单 8-11。

代码清单 8-11 OS_TS_GET()函数

```
 1 /*
 2 ************************************************************************
 3 *                                 时间戳配置
 4 ************************************************************************
 5 */
 6 /* 使能时间戳，在 os_cfg.h 头文件中使能 */
 7 #define OS_CFG_TS_EN                         1u
 8
 9 #if      OS_CFG_TS_EN == 1u
10 #define  OS_TS_GET()                (CPU_TS)CPU_TS_TmrRd()
11 #else
12 #define  OS_TS_GET()                (CPU_TS)0u
13 #endif
```

8.4 main 函数

主函数与上一章区别不大，首先在 main 函数开头加入 CPU_Init()函数，然后在任务 1 中对延时函数的执行时间进行测量。新加入的代码做了加粗显示，具体见代码清单 8-12。

代码清单 8-12 主函数

```
 1 uint32_t TimeStart;          /* 定义三个全局变量 */
 2 uint32_t TimeEnd;
 3 uint32_t TimeUse;
 4
 5
 6 /*
 7 ************************************************************************
 8 *                                 main 函数
 9 ************************************************************************
10 */
11
12 int main(void)
13 {
14     OS_ERR err;
15
16
17     /* CPU 初始化：1、初始化时间戳 */
18     CPU_Init();
19
20     /* 关闭中断 */
21     CPU_IntDis();
22
23     /* 配置 SysTick 10ms 中断一次 */
24     OS_CPU_SysTickInit (10);
25
26     /* 初始化相关的全局变量 */
27     OSInit(&err);
28
```

```
29      /* 创建任务 */
30      OSTaskCreate ((OS_TCB*)       &Task1TCB,
31                    (OS_TASK_PTR ) Task1,
32                    (void *)        0,
33                    (CPU_STK*)      &Task1Stk[0],
34                    (CPU_STK_SIZE) TASK1_STK_SIZE,
35                    (OS_ERR *)      &err);
36
37      OSTaskCreate ((OS_TCB*)       &Task2TCB,
38                    (OS_TASK_PTR ) Task2,
39                    (void *)        0,
40                    (CPU_STK*)      &Task2Stk[0],
41                    (CPU_STK_SIZE) TASK2_STK_SIZE,
42                    (OS_ERR *)      &err);
43
44      /* 将任务加入到就绪列表 */
45      OSRdyList[0].HeadPtr = &Task1TCB;
46      OSRdyList[1].HeadPtr = &Task2TCB;
47
48      /* 启动 OS，将不再返回 */
49      OSStart(&err);
50  }
51
52  /* 任务 1 */
53  void Task1( void *p_arg )
54  {
55      for ( ;; ) {
56          flag1 = 1;
57
58          TimeStart = OS_TS_GET();
59          OSTimeDly(20);
60          TimeEnd = OS_TS_GET();
61          TimeUse = TimeEnd - TimeStart;
62
63          flag1 = 0;
64          OSTimeDly(2);
65      }
66  }
```

8.5 实验想象

时间戳时间测量功能在软件仿真的时候使用不了，只能硬件仿真，这里仅能够讲解代码功能。有关硬件仿真，本书有提供一个测量 SysTick 定时时间的例程，名称叫“7-SysTick—系统定时器 STM32 时间戳【硬件仿真】”，在配套的程序源码里面可以找到。

第9章 临界段

9.1 临界段简介

9.2 关中断

9.2.1 测量关中断时间

9.3 锁调度器

9.3.1 测量锁调度器时间

9.4 main 函数

本章 main 函数没有添加新的测试代码，只需理解章节内容即可。

9.5 实验现象

本章没有实验，只需理解章节内容即可。

第10章　就绪列表

在 uC/OS-III 中，任务被创建后，任务的 TCB 会被放入就绪列表中，表示任务在就绪，随时可能被运行。就绪列表包含一个表示任务优先级的优先级表，一个存储任务 TCB 的 TCB 双向链表。

10.1 优先级表

优先级表在代码层面上来看，就是一个数组，在文件 os_prio.c（os_prio.c 第一次使用需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III Source 组）的开头定义，具体见代码清单 10-1。

代码清单 10-1 优先级表 OSPrioTbl[]定义

```
1 /* 定义优先级表，在 os.h 中用 extern 声明 */
2 CPU_DATA    OSPrioTbl[OS_PRIO_TBL_SIZE];                    (1)
```

代码清单 10-1（1）：正如我们所说，优先级表是一个数组，数组类型为 CPU_DATA，在 Cortex-M 内核芯片的 MCU 中 CPU_DATA 为 32 位整型。数组的大小由宏 OS_PRIO_TBL_SIZE 控制。OS_PRIO_TBL_SIZE 的具体取值与 uC/OS-III 支持多少个优先级有关，支持的优先级越多，优先级表也就越大，需要的 RAM 空间也就越多。理论上 uC/OS-III 支持无限的优先级，只要 RAM 控制足够。宏 OS_PRIO_TBL_SIZE 在 os.h 文件定义，具体实现见代码清单 10-2。

代码清单 10-2 OS_PRIO_TBL_SIZE 宏定义

```
1                                          (1)                    (2)
2 #define  OS_PRIO_TBL_SIZE    ((OS_CFG_PRIO_MAX - 1u) / (DEF_INT_CPU_NBR_BITS) + 1u)
```

代码清单 10-2（1）：OS_CFG_PRIO_MAX 表示支持多少个优先级，在 os_cfg.h 中定义，本书设置为 32，即最大支持 32 个优先级。

代码清单 10-2（2）：DEF_INT_CPU_NBR_BITS 定义 CPU 整型数据有多少位，本书适配的是基于 Cortex-M 系列的 MCU，宏展开为 32 位。

所以，经过 OS_CFG_PRIO_MAX 和 DEF_INT_CPU_NBR_BITS 这两个宏展开运算之后，可得出 OS_PRIO_TBL_SIZE 的值为 1，即优先级表只需要一个成员即可表示 32 个优先级。如果要支持 64 个优先级，即需要两个成员，以此类推。如果 MCU 的类型是 16 位、8 位或者 64 位，只需要把优先级表的数据类型 CPU_DATA 改成相应的位数即可。

那么优先级表又是如何跟任务的优先级联系在一起的？具体的优先级表的示意图见图 10-1。

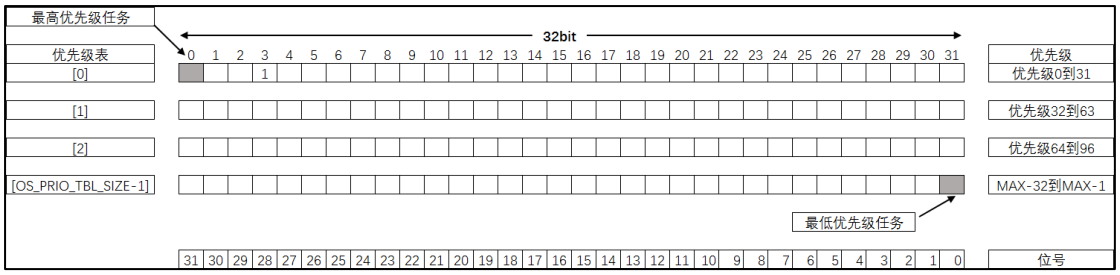


图 10-1 优先级表

在图 10-1 中，优先级表的成员是 32 位的，每个成员可以表示 32 个优先级。如果优先级超过 32 个，那么优先级表的成员就要相应的增加。以本书为例，CPU 的类型为 32 位，支持最大的优先级为 32 个，优先级表只需要一个成员即可，即只有 OSPrioTbl[0]。假如创建一个优先级为 Prio 的任务，那么就在 OSPrioTbl[0]的位[31-prio]置 1 即可。如果 Prio 等于 3，那么就将位 28 置 1。OSPrioTbl[0]的位 31 表示的是优先级最高的任务，以此递减，直到 OSPrioTbl[OS_PRIO_TBL_SIZE-1]]的位 0，OSPrioTbl[OS_PRIO_TBL_SIZE-1]]的位 0 表示的是最低的优先级。

10.1.1 优先级表函数讲解

优先级表相关的函数在 os_prio.c 文件中实现，在 os.h 文件中声明，函数汇总具体见表格 10-1。

表格 10-1 优先级表相关函数汇总

函数名称	函数作用
OS_PrioInit	初始化优先级表
OS_PrioInsert	设置优先级表中相应的位
OS_PrioRemove	清除优先级表中相应的位
OS_PrioGetHighest	查找最高的优先级

1. OS_PrioInit()函数

OS_PrioInit()函数用于初始化优先级表，在 OSInit()函数中被调用，具体实现见代码清单 10-3。

代码清单 10-3 OS_PrioInit()函数

```
1  /* 初始化优先级表 */
2  void OS_PrioInit( void )
3  {
4      CPU_DATA i;
5
6      /* 默认全部初始化为 0 */
7      for ( i=0u; i<OS_PRIO_TBL_SIZE; i++ ) {
8          OSPrioTbl[i] = (CPU_DATA)0;
9      }
10 }
```

本书中，优先级表 OS_PrioTbl[]只有一个成员，即 OS_PRIO_TBL_SIZE 等于 1 经过代码清单 10-3 初始化之后，具体示意图见图 10-2。

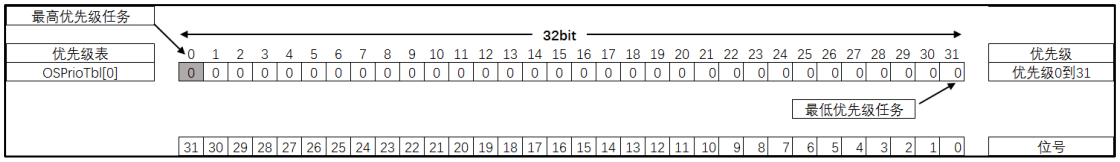


图 10-2 优先级表初始化后的示意图

2. OS_PrioInsert()函数

OS_PrioInsert()函数用于置位优先级表中相应的位，会被 OSTaskCreate()函数调用，具体实现见代码清单 10-4。

代码清单 10-4 OS_PrioInsert()函数

```
1  /* 置位优先级表中相应的位 */
2  void  OS_PrioInsert (OS_PRIO  prio)
3  {
4      CPU_DATA  bit;
5      CPU_DATA  bit_nbr;
6      OS_PRIO   ix;
7
8
9      /* 求模操作，获取优先级表数组的下标索引 */
10     ix              = prio / DEF_INT_CPU_NBR_BITS;                    (1)
11
12     /* 求余操作，将优先级限制在 DEF_INT_CPU_NBR_BITS 之内 */
13     bit_nbr         = (CPU_DATA)prio & (DEF_INT_CPU_NBR_BITS - 1u);   (2)
14
15     /* 获取优先级在优先级表中对应的位的位置 */                          (3)
16     bit             = 1u;
17     bit           <<= (DEF_INT_CPU_NBR_BITS - 1u) - bit_nbr;
18
19     /* 将优先级在优先级表中对应的位置 1 */
20     OSPrioTbl[ix] |= bit;                                             (4)
21 }
```

代码清单 10-4（1）：求模操作，获取优先级表数组的下标索引。即定位 prio 这个优先级对应优先级表数组的哪个成员。假设 prio 等于 3，DEF_INT_CPU_NBR_BITS（用于表示 CPU 一个整型数有多少位）等于 32，那么 ix 就等于 0，即对应 OSPrioTBL[0]。

代码清单 10-4（2）：求余操作，将优先级限制在 DEF_INT_CPU_NBR_BITS 之内，超过 DEF_INT_CPU_NBR_BITS 的优先级就肯定要增加优先级表的数组成员了。假设 prio 等于 3，DEF_INT_CPU_NBR_BITS（用于表示 CPU 一个整型数有多少位）等于 32，那么 bit_nbr 就等于 3，但是这个还不是真正需要被置位的位。

代码清单 10-4（3）：获取优先级在优先级表中对应的位的位置。置位优先级对应的位是从高位开始的，不是从低位开始。位 31 对应的是优先级 0，在 uC/OS-III 中，优先级数值越小，逻辑优先级就越高。假设 prio 等于 3，DEF_INT_CPU_NBR_BITS（用于表示 CPU 一个整型数有多少位）等于 32，那么 bit 就等于 28。

代码清单 10-4（4）：将优先级在优先级表中对应的位置 1。假设 prio 等于 3，DEF_INT_CPU_NBR_BITS（用于表示 CPU 一个整型数有多少位）等于 32，那么置位的就是 OSPrioTbl[0]的位 28。

在优先级最大是 32，DEF_INT_CPU_NBR_BITS 等于 32 的情况下，如果分别创建了优先级 3、5、8 和 11 这四个任务，任务创建成功后，优先级表的设置情况是怎么样的？具体见图 10-3。有一点要注意的是，在 uC/OS-III 中，最高优先级和最低优先级是留给系统任务使用的，用户任务不能使用。

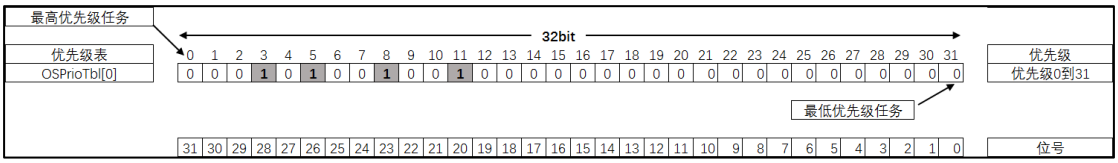


图 10-3 创建优先级 3、5、8 和 11 后优先级表的设置情况

3. OS_PrioRemove()函数

OS_PrioRemove()函数用于清除优先级表中相应的位，与 OS_PrioInsert()函数的作用刚好相反，具体实现见**错误!未找到引用源。**，有关代码的讲解参考代码清单 10-4 即可，不同的是置位操作改成了清 0。

代码清单 10-5 OS_PrioRemove()函数

```
1  /* 清除优先级表中相应的位 */
2  void  OS_PrioRemove (OS_PRIO  prio)
3  {
4      CPU_DATA  bit;
5      CPU_DATA  bit_nbr;
6      OS_PRIO   ix;
7
8
9      /* 求模操作，获取优先级表数组的下标索引 */
10     ix              = prio / DEF_INT_CPU_NBR_BITS;
11
12     /* 求余操作，将优先级限制在 DEF_INT_CPU_NBR_BITS 之内 */
13     bit_nbr         = (CPU_DATA)prio & (DEF_INT_CPU_NBR_BITS - 1u);
14
15     /* 获取优先级在优先级表中对应的位的位置 */
16     bit             = 1u;
17     bit           <<= (DEF_INT_CPU_NBR_BITS - 1u) - bit_nbr;
18
19     /* 将优先级在优先级表中对应的位清 0 */
20     OSPrioTbl[ix] &= ~bit;
21 }
```

4. OS_PrioGetHighest()函数

OS_PrioGetHighest()函数用于从优先级表中查找最高的优先级，具体实现见代码清单 10-6。

代码清单 10-6 OS_PrioGetHighest()函数

```
1  /* 获取最高的优先级 */
2  OS_PRIO  OS_PrioGetHighest (void)
3  {
4      CPU_DATA  *p_tbl;
5      OS_PRIO    prio;
```

```
 6
 7
 8      prio  = (OS_PRIO)0;
 9      /* 获取优先级表首地址 */
10      p_tbl = &OSPrioTbl[0];                                    (1)
11
12      /* 找到数值不为 0 的数组成员 */                              (2)
13      while (*p_tbl == (CPU_DATA)0) {
14          prio += DEF_INT_CPU_NBR_BITS;
15          p_tbl++;
16      }
17
18      /* 找到优先级表中置位的最高的优先级 */
19      prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);                (3)
20      return (prio);
21  }
```

代码清单 10-6（1）：获取优先级表的受地址，从头开始搜索整个优先级表，直到找到最高的优先级。

代码清单 10-6（2）：找到优先级表中数值不为 0 的数组成员，只要不为 0 就表示该成员里面至少有一个位是置位的。我们知道，在图 10-4 的优先级表中，优先级按照从左到右，从上到下依次减小，左上角为最高的优先级，右下角为最低的优先级，所以我们只需要找到第一个不是 0 的优先级表成员即可。

代码清单 10-6（3）：确定好优先级表中第一个不为 0 的成员后，然后再找出该成员中第一个置 1 的位（从高位到低位开始找）就算找到最高优先级。在一个变量中，按照从高位到低位的顺序查找第一个置 1 的位的方法是通过计算前导 0 函数 CPU_CntLeadZeros()来实现的。从高位开始找 1 叫计算前导 0，从低位开始找 1 叫计算后导 0。如果分别创建了优先级 3、5、8 和 11 这四个任务，任务创建成功后，优先级表的设置情况具体见图 10-5。调用 CPU_CntLeadZeros()可以计算出 OSPrioTbl[0]第一个置 1 的位前面有 3 个 0，那么这个 3 就是我们要查找的最高优先级，至于后面还有多少个位置 1 我们都不用管，只需要找到第一个 1 即可。

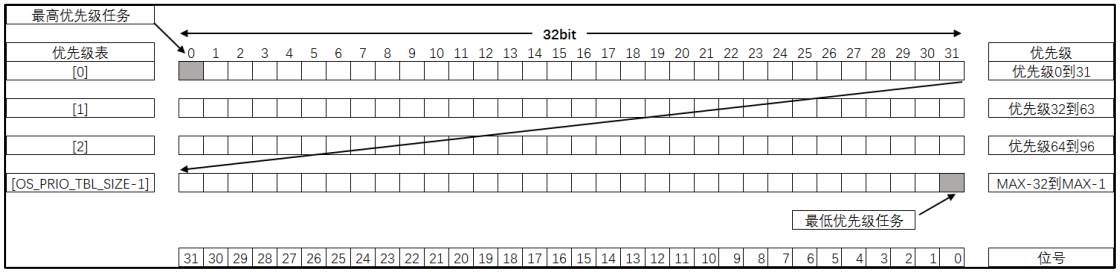


图 10-4 优先级表

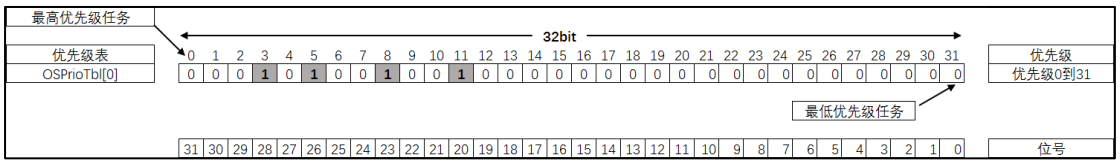


图 10-5 创建优先级 3、5、8 和 11 后优先级表的设置情况

CPU_CntLeadZeros()函数可由汇编或者 C 来实现，如果使用的处理器支持前导零指令 CLZ，可由汇编来实现，加快指令运算，如果不支持则由 C 来实现。在 uC/OS-III 中，这两种实现方法均有提供代码，到底使用哪种方法由 CPU_CFG_LEAD_ZEROS_ASM_PRESEN 这个宏来控制，定义了这个宏则使用汇编来实现，没有定义则使用 C 来实现。

Cortex-M 系列处理器自带 CLZ 指令，所以 CPU_CntLeadZeros()函数默认由汇编编写，具体在 cpu_a.asm 文件实现，在 cpu.h 文件声明，具体见代码清单 10-7。

代码清单 10-7 CPU_CntLeadZeros()函数实现与声明

```
;*************************************************************************
;                                 PUBLIC FUNCTIONS
;*************************************************************************
        EXPORT  CPU_CntLeadZeros
        EXPORT  CPU_CntTrailZeros

;*************************************************************************
;                                计算前导 0 函数
;
; 描述       :
;
; 函数声明   :  CPU_DATA  CPU_CntLeadZeros(CPU_DATA  val);
;
;*************************************************************************
CPU_CntLeadZeros
        CLZ     R0, R0                          ; Count leading zeros
        BX      LR



;*************************************************************************
;                                计算后导 0 函数
;
; 描述       :
;
; 函数声明   :  CPU_DATA  CPU_CntTrailZeros(CPU_DATA  val);
;
;*************************************************************************

CPU_CntTrailZeros
        RBIT    R0, R0                          ; Reverse bits
        CLZ     R0, R0                          ; Count trailing zeros
        BX      LR
```

```
/*
*************************************************************************
*                                函数声明
*                              cpu.h 文件
*************************************************************************
*/
#define     CPU_CFG_LEAD_ZEROS_ASM_PRESEN
CPU_DATA    CPU_CntLeadZeros (CPU_DATA    val);    /* 在 cpu_a.asm 定义 */
CPU_DATA    CPU_CntTrailZeros(CPU_DATA  val);      /* 在 cpu_a.asm 定义 */
```

如果处理器不支持前导 0 指令，CPU_CntLeadZeros()函数就得由 C 编写，具体在 cpu_core.c 文件实现，在 cpu.h 文件声明，具体见代码清单 10-8。

代码清单 10-8 由 C 实现的 CPU_CntLeadZeros()函数

```
#ifndef   CPU_CFG_LEAD_ZEROS_ASM_PRESENT
CPU_DATA  CPU_CntLeadZeros (CPU_DATA  val)
```

```
 3 {
 4     CPU_DATA    nbr_lead_zeros;
 5     CPU_INT08U  ix;
 6
 7     /* 检查高 16 位 */
 8     if (val > 0x0000FFFFu) {                                    (1)
 9         /* 检查 bits [31:24] : */
10         if (val > 0x00FFFFFFu) {                                (2)
11
12             /* 获取 bits [31:24]的值，并转换成 8 位 */
13             ix             = (CPU_INT08U)(val >> 24u);             (3)
14             /* 查表找到优先级 */
15             nbr_lead_zeros=(CPU_DATA)(CPU_CntLeadZerosTbl[ix]+0u); (4)
16
17         }
18         /* 检查 bits [23:16] : */
19         else {
20             /* 获取 bits [23:16]的值，并转换成 8 位 */
21             ix             = (CPU_INT08U)(val >> 16u);
22             /* 查表找到优先级 */
23             nbr_lead_zeros = (CPU_DATA  )(CPU_CntLeadZerosTbl[ix] +  8u);
24         }
25
26     }
27     /* 检查低 16 位 */
28     else {
29         /* 检查 bits [15:08] : */
30         if (val > 0x000000FFu) {
31             /* 获取 bits [15:08]的值，并转换成 8 位 */
32             ix             = (CPU_INT08U)(val >>  8u);
33             /* 查表找到优先级 */
34             nbr_lead_zeros = (CPU_DATA  )(CPU_CntLeadZerosTbl[ix] + 16u);
35
36         }
37         /* 检查 bits [07:00] : */
38         else {
39             /* 获取 bits [15:08]的值，并转换成 8 位 */
40             ix             = (CPU_INT08U)(val >>  0u);
41             /* 查表找到优先级 */
42             nbr_lead_zeros = (CPU_DATA  )(CPU_CntLeadZerosTbl[ix] + 24u);
43         }
44     }
45
46     /* 返回优先级 */
47     return (nbr_lead_zeros);
48 }
49 #endif
```

在 uC/OS-III 中，由 C 实现的 CPU_CntLeadZeros()函数支持 8 位、16 位、32 位和 64 位的变量的前导 0 计算，但最终的代码实现都是分离成 8 位来计算。这里我们只讲解 32 位的，其它几种情况都类似。

代码清单 10-8（1）：分离出高 16 位，else 则为低 16 位。

代码清单 10-8（2）：分离出高 16 位的高 8 位，else 则为高 16 位的低 8 位。

代码清单 10-8（3）：将高 16 位的高 8 位通过移位强制转化为 8 位的变量，用于后面的查表操作。

代码清单 10-8（4）：将 8 位的变量 ix 作为数组 CPU_CntLeadZerosTbl[]的索引，返回索引对应的值，那么该值就是 8 位变量 ix 对应的前导 0，然后再加上（24-右移的位数）就等于优先级。数组 CPU_CntLeadZerosTbl[]在 cpu_core.c 的开头定义，具体见代码清单 10-9。

代码清单 10-9 CPU_CntLeadZerosTbl[]定义

```
#ifndef   CPU_CFG_LEAD_ZEROS_ASM_PRESENT
static  const  CPU_INT08U  CPU_CntLeadZerosTbl[256] = {/*   索引           */
    8u,7u,6u,6u,5u,5u,5u,5u,4u,4u,4u,4u,4u,4u,4u,4u,    /*   0x00 to 0x0F   */
    3u,3u,3u,3u,3u,3u,3u,3u,3u,3u,3u,3u,3u,3u,3u,3u,    /*   0x10 to 0x1F   */
    2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,    /*   0x20 to 0x2F   */
    2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,2u,    /*   0x30 to 0x3F   */
    1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,    /*   0x40 to 0x4F   */
    1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,    /*   0x50 to 0x5F   */
    1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,    /*   0x60 to 0x6F   */
    1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,1u,    /*   0x70 to 0x7F   */
    0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,    /*   0x80 to 0x8F   */
    0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,    /*   0x90 to 0x9F   */
    0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,    /*   0xA0 to 0xAF   */
    0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,    /*   0xB0 to 0xBF   */
    0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,    /*   0xC0 to 0xCF   */
    0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,    /*   0xD0 to 0xDF   */
    0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,    /*   0xE0 to 0xEF   */
    0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u,0u     /*   0xF0 to 0xFF   */
};
#endif
```

代码清单 10-8 中，对一个 32 位的变量算前导 0 个数的时候都是分离成 8 位的变量来计算，然后将这个 8 位的变量作为数组 CPU_CntLeadZerosTbl[]的索引，索引下对应的值就是这个 8 位变量的前导 0 个数。一个 8 位的变量的取值范围为 0~0XFF，这些值作为数组 CPU_CntLeadZerosTbl[]的索引，每一个值的前导 0 个数都预先算出来作为该数组索引下的值。通过查 CPU_CntLeadZerosTbl[]这个表就可以很快的知道一个 8 位变量的前导 0 个数，根本不用计算，只是浪费了定义 CPU_CntLeadZerosTbl[]这个表的一点点空间而已，在处理器内存很充足的情况下，则优先选择这种空间换时间的方法。

10.2 就绪列表

准备好运行的任务的 TCB 都会被放到就绪列表中，系统可随时调度任务运行。就绪列表在代码的层面上看就是一个 OS_RDY_LIST 数据类型的数组 OSRdyList[]，数组的大小由宏 OS_CFG_PRIO_MAX 决定，支持多少个优先级，OSRdyList[]就有多少个成员。任务的优先级与 OSRdyList[]的索引一一对应，比如优先级 3 的任务的 TCB 会被放到 OSRdyList[3]中。OSRdyList[]是一个在 os.h 文件中定义的全局变量，具体见代码清单 10-10。

代码清单 10-10 OSRdyList[]数组定义

```
/* 就绪列表定义 */
OS_EXT    OS_RDY_LIST    OSRdyList[OS_CFG_PRIO_MAX];
```

代码清单 10-10 中的数据类型 OS_RDY_LIST 在 os.h 中定义，专用于就绪列表，具体实现见代码清单 10-11。

代码清单 10-11 OS_RDY_LIST 定义

```
typedef  struct  os_rdy_list          OS_RDY_LIST;  (1)

struct os_rdy_list {
    OS_TCB        *HeadPtr;                        (2)
    OS_TCB        *TailPtr;
    OS_OBJ_QTY    NbrEntries;                      (3)
};
```

代码清单 10-11（1）：在 uC/OS-III 中，内核对象的数据类型都会用大写字母重新定义。

代码清单 10-11（2）：OSRdyList[]的成员与任务的优先级一一对应，同一个优先级的多个任务会以双向链表的形式存在 OSRdyList[]同一个索引下，那么 HeadPtr 就用于指向链表的头节点，TailPtr 用于指向链表的尾节点，该优先级下的索引成员的地址则称为该优先级下双向链表的根节点，知道根节点的地址就可以查找到该链表下的每一个节点。

代码清单 10-11（3）：NbrEntries 表示 OSRdyList[]同一个索引下有多少个任务。

一个空的就绪列表，OSRdyList[]索引下的 HeadPtr、TailPtr 和 NbrEntrie 都会被初始化为 0，具体见图 10-6。

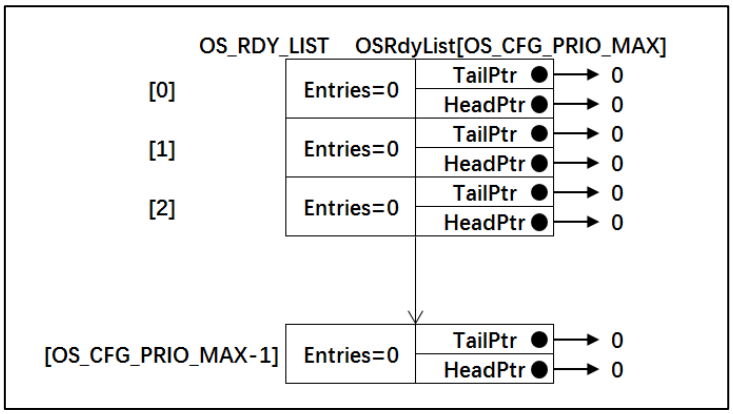


图 10-6 空的就绪列表

就绪列表相关的所有函数都在 os_core.c 实现，这些函数都是以“OS_”开头，表示是 OS 的内部函数，用户不能调用，这些函数的汇总具体见表格 10-2。

表格 10-2 就绪列表相关函数汇总

函数名称	函数作用
OS_RdyListInit	初始化就绪列表为空
OS_RdyListInsert	插入一个 TCB 到就绪列表
OS_RdyListInsertHead	插入一个 TCB 到就绪列表的头部
OS_RdyListInsertTail	插入一个 TCB 到就绪列表的尾部
OS_RdyListMoveHeadToTail	将 TCB 从就绪列表的头部移到尾部
OS_RdyListRemove	将 TCB 从就绪列表中移除

10.2.1 就绪列表函数讲解

在实现就绪列表相关函数之前，我们需要在结构体 os_tcb 中添加 Prio、NextPtr 和 PrevPtr 这三个成员，然后在 os.h 中定义两个全局变量 OSPrioCur 和 OSPrioHighRdy，具体定义见代码清单 10-12。接下来要实现的就绪列表相关的函数会用到几个变量。

代码清单 10-12 就绪列表函数需要用到的变量定义

```
1 struct os_tcb {
2     CPU_STK         *StkPtr;
3     CPU_STK_SIZE    StkSize;
4 
5     /* 任务延时周期个数 */
```

```
    OS_TICK         TaskDelayTicks;

    /* 任务优先级 */
    OS_PRIO         Prio;

    /* 就绪列表双向链表的下一个指针 */
    OS_TCB          *NextPtr;
    /* 就绪列表双向链表的前一个指针 */
    OS_TCB          *PrevPtr;
};

/* 在 os.h 中定义 */
OS_EXT    OS_PRIO  OSPrioCur;        /* 当前优先级 */
OS_EXT    OS_PRIO  OSPrioHighRdy;    /* 最高优先级 */
```

1. OS_RdyListInit()函数

OS_RdyListInit()用于将就绪列表 OSRdyList[]初始化为空，初始化完毕之后具体示意图见图 10-6，具体实现见代码清单 10-13。

代码清单 10-13 OS_RdyListInit()函数

```
void OS_RdyListInit(void)
{
    OS_PRIO i;
    OS_RDY_LIST *p_rdy_list;

    /* 循环初始化，所有成员都初始化为 0 */
    for ( i=0u; i<OS_CFG_PRIO_MAX; i++ ) {
        p_rdy_list = &OSRdyList[i];
        p_rdy_list->NbrEntries = (OS_OBJ_QTY)0;
        p_rdy_list->HeadPtr = (OS_TCB *)0;
        p_rdy_list->TailPtr = (OS_TCB *)0;
    }
}
```

2. OS_RdyListInsertHead()函数

OS_RdyListInsertHead()用于在链表头部插入一个 TCB 节点，插入的时候分两种情况，第一种是链表是空链表，第二种是链表中已有节点，具体示意图见图 10-7，具体的代码实现见代码清单 10-14，阅读代码的时候最好配套示意图来理解。

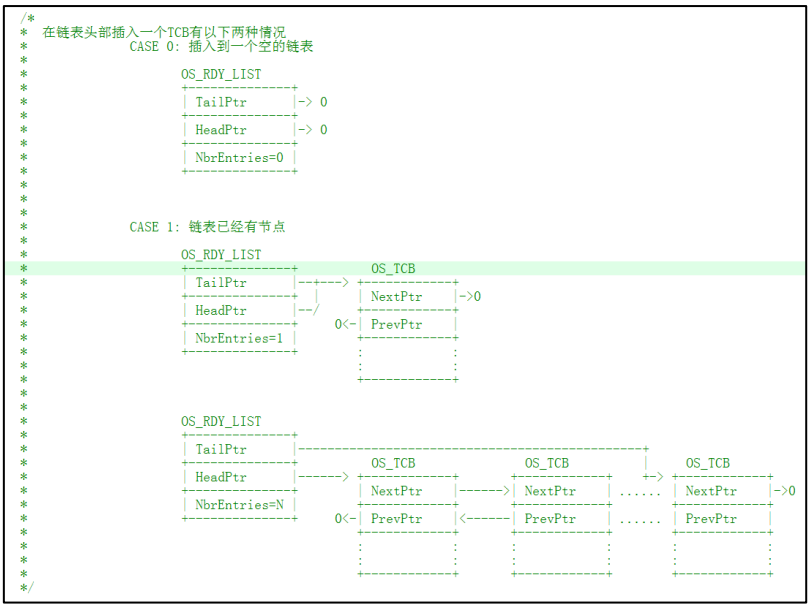


图 10-7 在链表的头部插入一个 TCB 节点前链表的可能情况

代码清单 10-14 OS_RdyListInsertHead()函数

```
1  void  OS_RdyListInsertHead (OS_TCB  *p_tcb)
2  {
3      OS_RDY_LIST  *p_rdy_list;
4      OS_TCB       *p_tcb2;
5
6
7
8      /* 获取链表根部 */
9      p_rdy_list = &OSRdyList[p_tcb->Prio];
10
11     /* CASE 0: 链表是空链表 */
12     if (p_rdy_list->NbrEntries == (OS_OBJ_QTY)0) {
13         p_rdy_list->NbrEntries =  (OS_OBJ_QTY)1;
14         p_tcb->NextPtr         =  (OS_TCB    *)0;
15         p_tcb->PrevPtr         =  (OS_TCB    *)0;
16         p_rdy_list->HeadPtr    =  p_tcb;
17         p_rdy_list->TailPtr    =  p_tcb;
18     }
19     /* CASE 1: 链表已有节点 */
20     else {
21         p_rdy_list->NbrEntries++;
22         p_tcb->NextPtr         = p_rdy_list->HeadPtr;
23         p_tcb->PrevPtr         = (OS_TCB     *)0;
24         p_tcb2                 = p_rdy_list->HeadPtr;
25         p_tcb2->PrevPtr        = p_tcb;
26         p_rdy_list->HeadPtr    = p_tcb;
27     }
28 }
```

3. OS_RdyListInsertTail()函数

OS_RdyListInsertTail()用于在链表尾部插入一个 TCB 节点，插入的时候分两种情况，第一种是链表是空链表，第二种是链表中已有节点，具体示意图见图 10-8，具体的代码实现见，阅读代码的时候最好配套示意图来理解。

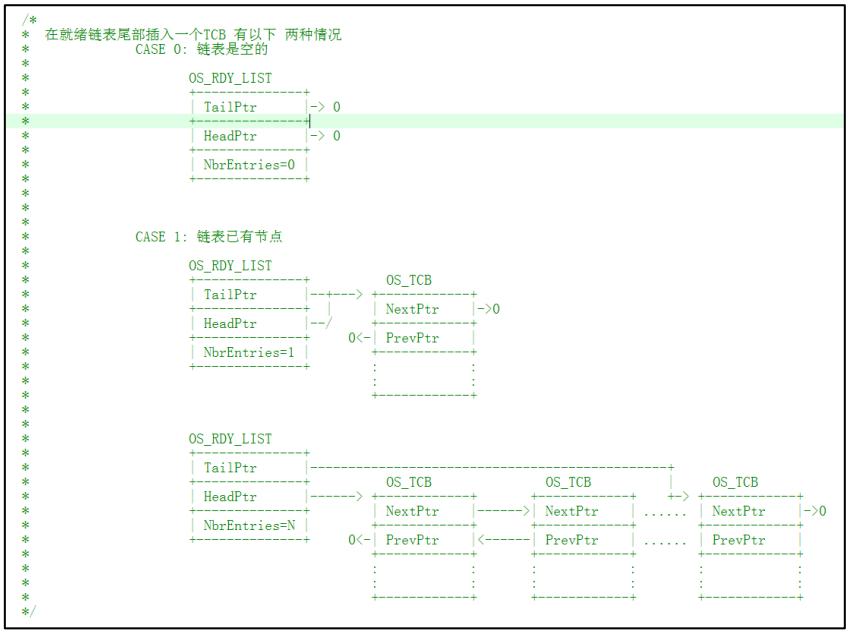


图 10-8 在链表的尾部插入一个 TCB 节点前链表的可能情况

代码清单 10-15 OS_RdyListInsertTail()函数

```
1  void  OS_RdyListInsertTail (OS_TCB  *p_tcb)
2  {
3      OS_RDY_LIST  *p_rdy_list;
4      OS_TCB       *p_tcb2;
5
6
7      /* 获取链表根部 */
8      p_rdy_list = &OSRdyList[p_tcb->Prio];
9
10     /* CASE 0: 链表是空链表 */
11     if (p_rdy_list->NbrEntries == (OS_OBJ_QTY)0) {
12         p_rdy_list->NbrEntries  = (OS_OBJ_QTY)1;
13         p_tcb->NextPtr          = (OS_TCB   *)0;
14         p_tcb->PrevPtr          = (OS_TCB   *)0;
15         p_rdy_list->HeadPtr     = p_tcb;
16         p_rdy_list->TailPtr     = p_tcb;
17     }
18     /* CASE 1: 链表已有节点 */
19     else {
20         p_rdy_list->NbrEntries++;
21         p_tcb->NextPtr          = (OS_TCB   *)0;
22         p_tcb2                  = p_rdy_list->TailPtr;
23         p_tcb->PrevPtr          = p_tcb2;
24         p_tcb2->NextPtr         = p_tcb;
25         p_rdy_list->TailPtr     = p_tcb;
```

```
26      }
27  }
```

4. OS_RdyListInsert()函数

OS_RdyListInsert()用于将任务的 TCB 插入到就绪列表，插入的时候分成两步，第一步是根据优先级将优先级表中的相应位置位，这个调用 OS_PrioInsert()函数来实现，第二步是根据优先级将任务的 TCB 放到 OSRdyList[优先级]中，如果优先级等于当前的优先级则插入到链表的尾部，否则插入到链表的头部，具体实现见代码清单 10-16。

代码清单 10-16 OS_RdyListInsert()函数

```
 1 /* 在就绪链表中插入一个 TCB */
 2 void  OS_RdyListInsert (OS_TCB  *p_tcb)
 3 {
 4     /* 将优先级插入到优先级表 */
 5     OS_PrioInsert(p_tcb->Prio);
 6
 7     if (p_tcb->Prio == OSPrioCur) {
 8         /* 如果是当前优先级则插入到链表尾部 */
 9         OS_RdyListInsertTail(p_tcb);
10     } else {
11         /* 否则插入到链表头部 */
12         OS_RdyListInsertHead(p_tcb);
13     }
14 }
```

5. OS_RdyListMoveHeadToTail()函数

OS_RdyListMoveHeadToTail()函数用于将节点从链表头部移动到尾部，移动的时候分四种情况，第一种是链表为空，无事可做；第二种是链表只有一个节点，也是无事可做；第三种是链表只有两个节点；第四种是链表有两个以上节点，具体示意图见图 10-9，具体代码实现见代码清单 10-17，阅读代码的时候最好配套示意图来理解。

```
/*
*  将链表中的节点从头部移动到尾部，包含下面三种情况：
*              CASE 0: 链表为空，无事可做。
*
*              CASE 1: 链表只有一个节点，无事可做。
*
*              CASE 2: 链表只有两个节点。
*
*                     OS_RDY_LIST
*                     +--------------+
*                     | TailPtr      |---------------------------+
*                     +--------------+                           |
*                     | HeadPtr      |------>  OS_TCB            |     OS_TCB
*                     +--------------+         +-------------+   +->  +-------------+
*                     | NbrEntries=2 |         | NextPtr     |------>  | NextPtr     |->0
*                     +--------------+         +-------------+         +-------------+
*                                         0<-| PrevPtr     | <------ | PrevPtr     |
*                                            +-------------+         +-------------+
*                                            :             :         :             :
*                                            :             :         :             :
*                                            +-------------+         +-------------+
*
*
*              CASE N: 链表有两个节点以上。
*
*                     OS_RDY_LIST
*                     +--------------+
*                     | TailPtr      |-----------------------------------------------+
*                     +--------------+                                               |
*                     | HeadPtr      |------>  OS_TCB            OS_TCB             |     OS_TCB
*                     +--------------+         +-------------+   +-------------+    +->  +-------------+
*                     | NbrEntries=N |         | NextPtr     |------>  NextPtr     | ..... | NextPtr     |->0
*                     +--------------+         +-------------+   +-------------+         +-------------+
*                                         0<-| PrevPtr     | <------ | PrevPtr     | ..... | PrevPtr     |
*                                            +-------------+         +-------------+       +-------------+
*                                            :             :         :             :       :             :
*                                            :             :         :             :       :             :
*                                            +-------------+         +-------------+       +-------------+
*/
```

图 10-9 将节点从链表头部移动到尾部前链表的可能情况

代码清单 10-17 OS_RdyListMoveHeadToTail()函数

```
 1 void  OS_RdyListMoveHeadToTail (OS_RDY_LIST  *p_rdy_list)
 2 {
 3     OS_TCB  *p_tcb1;
 4     OS_TCB  *p_tcb2;
 5     OS_TCB  *p_tcb3;
 6
 7
 8
 9     switch (p_rdy_list->NbrEntries) {
10     case 0:
11     case 1:
12         break;
13
14     case 2:
15         p_tcb1              = p_rdy_list->HeadPtr;
16         p_tcb2              = p_rdy_list->TailPtr;
17         p_tcb1->PrevPtr     = p_tcb2;
18         p_tcb1->NextPtr     = (OS_TCB *)0;
19         p_tcb2->PrevPtr     = (OS_TCB *)0;
20         p_tcb2->NextPtr     = p_tcb1;
21         p_rdy_list->HeadPtr = p_tcb2;
22         p_rdy_list->TailPtr = p_tcb1;
23         break;
24
25     default:
26         p_tcb1              = p_rdy_list->HeadPtr;
27         p_tcb2              = p_rdy_list->TailPtr;
28         p_tcb3              = p_tcb1->NextPtr;
29         p_tcb3->PrevPtr     = (OS_TCB *)0;
30         p_tcb1->NextPtr     = (OS_TCB *)0;
31         p_tcb1->PrevPtr     = p_tcb2;
32         p_tcb2->NextPtr     = p_tcb1;
33         p_rdy_list->HeadPtr = p_tcb3;
34         p_rdy_list->TailPtr = p_tcb1;
35         break;
36     }
37 }
```

6. OS_RdyListRemove()函数

OS_RdyListRemove()函数用于从链表中移除一个节点，移除的时候分为三种情况，第一种是链表为空，无事可做；第二种是链表只有一个节点；第三种是链表有两个以上节点，具体示意图见图 10-10，具体代码实现见，阅读代码的时候最好配套示意图来理解。

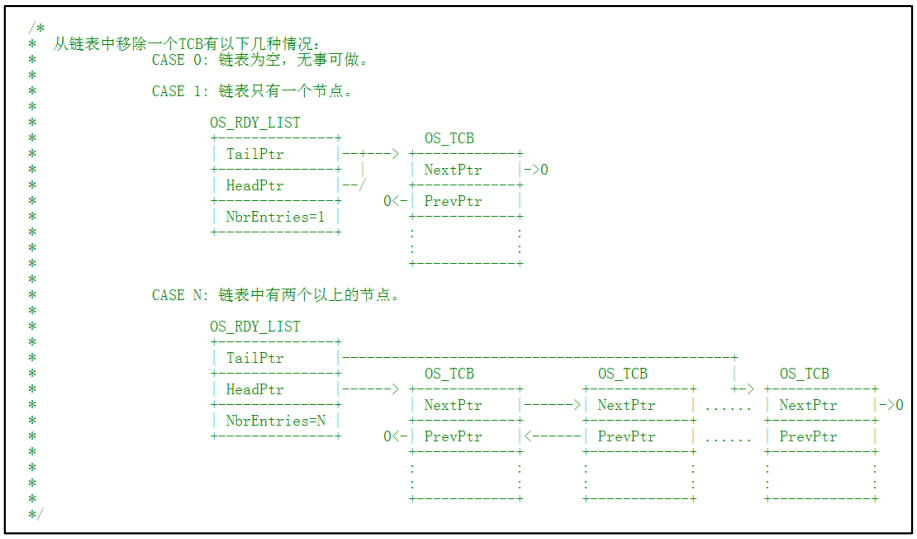


图 10-10 从链表中移除一个节点前链表的可能情况

代码清单 10-18 OS_RdyListRemove()函数

```
1  void  OS_RdyListRemove (OS_TCB  *p_tcb)
2  {
3      OS_RDY_LIST  *p_rdy_list;
4      OS_TCB       *p_tcb1;
5      OS_TCB       *p_tcb2;
6
7
8
9      p_rdy_list = &OSRdyList[p_tcb->Prio];
10
11     /* 保存要删除的 TCB 节点的前一个和后一个节点 */
12     p_tcb1     = p_tcb->PrevPtr;
13     p_tcb2     = p_tcb->NextPtr;
14
15     /* 要移除的 TCB 节点是链表中的第一个节点 */
16     if (p_tcb1 == (OS_TCB *)0) {
17         /* 且该链表中只有一个节点 */
18         if (p_tcb2 == (OS_TCB *)0) {
19             /* 根节点全部初始化为 0 */
20             p_rdy_list->NbrEntries = (OS_OBJ_QTY)0;
21             p_rdy_list->HeadPtr    = (OS_TCB    *)0;
22             p_rdy_list->TailPtr    = (OS_TCB    *)0;
23
24             /* 清除在优先级表中相应的位 */
25             OS_PrioRemove(p_tcb->Prio);
26         }
27         /* 该链表中不止一个节点 */
28         else {
29             /* 节点减 1 */
30             p_rdy_list->NbrEntries--;
31             p_tcb2->PrevPtr        = (OS_TCB    *)0;
32             p_rdy_list->HeadPtr    = p_tcb2;
33         }
34     }
35     /* 要移除的 TCB 节点不是链表中的第一个节点 */
36     else {
37         p_rdy_list->NbrEntries--;
38         p_tcb1->NextPtr = p_tcb2;
39
40         /* 如果要删除的节点的下一个节点是 0，即要删除的节点是最后一个节点 */
```

```
41          if (p_tcb2 == (OS_TCB *)0) {
42              p_rdy_list->TailPtr = p_tcb1;
43          } else {
44              p_tcb2->PrevPtr     = p_tcb1;
45          }
46      }
47
48      /* 复位从就绪列表中删除的 TCB 的 PrevPtr 和 NextPtr 这两个指针 */
49      p_tcb->PrevPtr = (OS_TCB *)0;
50      p_tcb->NextPtr = (OS_TCB *)0;
51  }
```

10.3 main 函数

本章 main 函数没有添加新的测试代码，只需理解章节内容即可。

10.4 实验现象

本章没有实验，只需理解章节内容即可。

第11章　支持多优先级

在本章之前，OS 还没有到优先级，只支持两个任务互相切换，从本章开始，任务中我们开始加入优先级的功能。在 uC/OS-III 中，数字优先级越小，逻辑优先级越高。

11.1 定义优先级相关全局变量

在支持任务多优先级的时候，需要在 os.h 头文件添加两个优先级相关的全局变量，具体定义见代码清单 11-1。

代码清单 11-1 定义优先级相关全局变量

```
1 /* 在 os.h 中定义 */
2 /* 当前优先级 */
3 OS_EXT             OS_PRIO                 OSPrioCur;
4 /* 最高优先级 */
5 OS_EXT             OS_PRIO                 OSPrioHighRdy;
```

11.2 修改 OSInit()函数

刚刚新添加的优先级相关的全部变量，需要在 OSInit()函数中进行初始化，具体见代码清单 11-2 中的加粗部分代码，其实 OS 中定义的所有的全局变量都是在 OSInit()中初始化的。

代码清单 11-2 OSInit()函数

```
1 void OSInit (OS_ERR *p_err)
2 {
3     /* 配置 OS 初始状态为停止态 */
4     OSRunning =  OS_STATE_OS_STOPPED;
5 
6     /* 初始化两个全局 TCB，这两个 TCB 用于任务切换 */
7     OSTCBCurPtr = (OS_TCB *)0;
8     OSTCBHighRdyPtr = (OS_TCB *)0;
9 
10    /* 初始化优先级变量 */
11    OSPrioCur                     = (OS_PRIO)0;
12    OSPrioHighRdy                 = (OS_PRIO)0;
13 
14    /* 初始化优先级表 */
15    OS_PrioInit();
16 
17    /* 初始化就绪列表 */
18    OS_RdyListInit();
19 
20    /* 初始化空闲任务 */
21    OS_IdleTaskInit(p_err);
22    if (*p_err != OS_ERR_NONE) {
23        return;
24    }
25 }
```

11.3 修改任务控制块 TCB

在任务控制块中，加入优先级字段 Prio，具体见代码清单 11-3 中的加粗代码。优先级 Prio 的数据类型为 OS_PRIO，宏展开后是 8 位的整型，所以只支持 255 个优先级。

代码清单 11-3 在 TCB 中加入优先级

```
1  struct os_tcb {
2      CPU_STK          *StkPtr;
3      CPU_STK_SIZE     StkSize;
4
5      /* 任务延时周期个数 */
6      OS_TICK          TaskDelayTicks;
7
8      /* 任务优先级 */
9      OS_PRIO          Prio;
10
11     /* 就绪列表双向链表的下一个指针 */
12     OS_TCB           *NextPtr;
13     /* 就绪列表双向链表的前一个指针 */
14     OS_TCB           *PrevPtr;
15 };
```

11.4 修改 OSTaskCreate()函数

修改 OSTaskCreate()函数，在里面加入优先级相关的处理，具体见代码清单 11-4 的加粗部分。

代码清单 11-4 OSTaskCreate()函数 加入优先级处理

```
1  void OSTaskCreate (OS_TCB        *p_tcb,
2                     OS_TASK_PTR   p_task,
3                     void          *p_arg,
4                     OS_PRIO       prio,                    (1)
5                     CPU_STK       *p_stk_base,
6                     CPU_STK_SIZE  stk_size,
7                     OS_ERR        *p_err)
8  {
9      CPU_STK       *p_sp;
10     CPU_SR_ALLOC();                                        (2)
11
12     /* 初始化 TCB 为默认值 */
13     OS_TaskInitTCB(p_tcb);                                 (3)
14
15     /* 初始化堆栈 */
16     p_sp = OSTaskStkInit( p_task,
17                           p_arg,
18                           p_stk_base,
19                           stk_size );
20
21     p_tcb->Prio = prio;                                    (4)
22
23     p_tcb->StkPtr = p_sp;
24     p_tcb->StkSize = stk_size;
25
26     /* 进入临界段 */
27     OS_CRITICAL_ENTER();                                   (5)
28
29     /* 将任务添加到就绪列表 */                              (6)
30     OS_PrioInsert(p_tcb->Prio);
31     OS_RdyListInsertTail(p_tcb);
32
```

```
33      /* 退出临界段 */
34      OS_CRITICAL_EXIT();                                             (7)
35
36      *p_err = OS_ERR_NONE;
37  }
```

代码清单 11-4（1）：在函数形参中，加入优先级字段。任务的优先级由用户在创建任务的时候通过形参 Prio 传进来。

代码清单 11-4（2）：定义一个局部变量，用来存 CPU 关中断前的中断状态，因为接下来将任务添加到就绪列表这段代码属于临界短代码，需要关中断。

代码清单 11-4（3）：初始化 TCB 为默认值，其实就是全部初始化为 0，OS_TaskInitTCB()函数在 os_task.c 的开头定义，具体见代码清单 11-5。

代码清单 11-5 OS_TaskInitTCB()函数

```
1  void  OS_TaskInitTCB (OS_TCB  *p_tcb)
2  {
3      p_tcb->StkPtr             = (CPU_STK       *)0;
4      p_tcb->StkSize            = (CPU_STK_SIZE   )0u;
5
6      p_tcb->TaskDelayTicks     = (OS_TICK       )0u;
7
8      p_tcb->Prio               = (OS_PRIO        )OS_PRIO_INIT;       (1)
9
10     p_tcb->NextPtr            = (OS_TCB        *)0;
11     p_tcb->PrevPtr            = (OS_TCB        *)0;
12 }
```

代码清单 11-5 （1）：OS_PRIO_INIT 是任务 TCB 初始化的时候给的默认的一个优先级，宏展开等于 OS_CFG_PRIO_MAX，这是一个不会被 OS 使用到的优先级。OS_PRIO_INIT 具体在 os.h 中定义。

代码清单 11-4（4）：将形参传进来的优先级存到任务控制块 TCB 的优先级字段。

代码清单 11-4（5）：进入临界段。

代码清单 11-4（6）：将任务插入到就绪列表，这里需要分成两步来实现：1、根据优先级置位优先级表中的相应位置；2、将任务 TCB 放到 OSRdyList[优先级]中，如果同一个优先级有多个任务，那么这些任务的 TCB 就会被放到 OSRdyList[优先级]串成一个双向链表。

代码清单 11-4（7）：退出临界段。

11.5 修改 OS_IdleTaskInit()函数

修改 OS_IdleTaskInit()函数，是因为该函数调用了任务创建函数 OSTaskCreate()，OSTaskCreate()我们刚刚加入了优先级，所以这里我们要跟空闲任务分配一个优先级，具体见。代码清单 11-6 的加粗部分。

代码清单 11-6 OS_IdleTaskInit()函数

```
1  /* 空闲任务初始化 */
2  void  OS_IdleTaskInit(OS_ERR  *p_err)
3  {
4      /* 初始化空闲任务计数器 */
5      OSIdleTaskCtr = (OS_IDLE_CTR)0;
6
```

```
 7      /* 创建空闲任务 */
 8      OSTaskCreate( (OS_TCB      *)&OSIdleTaskTCB,
 9                    (OS_TASK_PTR )OS_IdleTask,
10                    (void        *)0,
11                    (OS_PRIO)(OS_CFG_PRIO_MAX - 1u),                    (1)
12                    (CPU_STK     *)OSCfg_IdleTaskStkBasePtr,
13                    (CPU_STK_SIZE)OSCfg_IdleTaskStkSize,
14                    (OS_ERR      *)p_err );
15  }
```

代码清单 11-6（1）：空闲任务是 uC/OS-III 的内部任务，在 OSInit()中被创建，在系统没有任何用户任务运行的情况下，空闲任务就会被运行，优先级最低，即等于 OS_CFG_PRIO_MAX - 1u。

11.6 修改 OSStart()函数

加入优先级之后，OSStart()函数需要修改，具体哪一个任务最先运行，由优先级决定，新加入的代码具体见代码清单 11-7 的加粗部分。

代码清单 11-7 OSStart()函数

```
 1  /* 启动 RTOS，将不再返回 */
 2  void OSStart (OS_ERR *p_err)
 3  {
 4      if ( OSRunning == OS_STATE_OS_STOPPED ) {
 5  #if 0
 6          /* 手动配置任务 1 先运行 */
 7          OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
 8  #endif
 9          /* 寻找最高的优先级 */
10          OSPrioHighRdy   = OS_PrioGetHighest();                  (1)
11          OSPrioCur       = OSPrioHighRdy;
12
13          /* 找到最高优先级的 TCB */
14          OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;     (2)
15          OSTCBCurPtr     = OSTCBHighRdyPtr;
16
17          /* 标记 OS 开始运行 */
18          OSRunning       = OS_STATE_OS_RUNNING;
19
20          /* 启动任务切换，不会返回 */
21          OSStartHighRdy();
22
23          /* 不会运行到这里，运行到这里表示发生了致命的错误 */
24          *p_err = OS_ERR_FATAL_RETURN;
25      } else {
26          *p_err = OS_STATE_OS_RUNNING;
27      }
28  }
```

代码清单 11-7（1）：调取 OS_PrioGetHighest()函数从全局变量优先级表 OSPrioTbl[]获取最高的优先级，放到 OSPrioHighRdy 这个全局变量中，然后把 OSPrioHighRdy 的值再赋给当前优先级 OSPrioCur 这个全局变量。在任务切换的时候需要用到 OSPrioHighRdy 和 OSPrioCur 这两个全局变量。

代码清单 11-7（2）：根据 OSPrioHighRdy 的值，作为全局变量 OSRdyList[]的下标索引找到最高优先级任务的 TCB，传给全局变量 OSTCBHighRdyPtr，然后再将 OSTCBHighRdyPtr 赋值给 OSTCBCurPtr。在任务切换的时候需要使用到 OSTCBHighRdyPtr 和 OSTCBCurPtr 这两个全局变量。

11.7 修改 PendSV_Handler()函数

PendSV_Handler()函数中添加了优先级相关的代码，具体见代码清单 11-8 中加粗部分。有关 PendSV_Handler()这个函数的具体讲解要参考《任务的定义与任务切换的实现》这个章节，这里不再赘述。

代码清单 11-8 PendSV_Handler()函数

```
;*******************************************************************
;                                 PendSVHandler 异常
;*******************************************************************

OS_CPU_PendSVHandler_nosave

; OSPrioCur   = OSPrioHighRdy
    LDR      R0, =OSPrioCur
    LDR      R1, =OSPrioHighRdy
    LDRB     R2, [R1]
    STRB     R2, [R0]

; OSTCBCurPtr = OSTCBHighRdyPtr
  LDR      R0, = OSTCBCurPtr
  LDR      R1, = OSTCBHighRdyPtr
  LDR      R2, [R1]
  STR      R2, [R0]

  LDR      R0, [R2]
  LDMIA    R0!, {R4-R11}

  MSR      PSP, R0
  ORR      LR, LR, #0x04
  CPSIE    I
  BX       LR


  NOP

  ENDP
```

11.8 修改 OSTimeDly()函数

任务调用 OSTimeDly()函数之后，任务就处于阻塞态，需要将任务从就绪列表中移除，具体修改的代码见代码清单 11-9 的加粗部分。

代码清单 11-9 OSTimeDly()函数

```
/* 阻塞延时 */
void  OSTimeDly(OS_TICK dly)
{
#if 0
    /* 设置延时时间 */
    OSTCBCurPtr->TaskDelayTicks = dly;

    /* 进行任务调度 */
    OSSched();
#endif

    CPU_SR_ALLOC();                                                    (1)

    /* 进入临界区 */
```

```
15      OS_CRITICAL_ENTER();                                    (2)
16
17      /* 设置延时时间 */
18      OSTCBCurPtr->TaskDelayTicks = dly;
19
20      /* 从就绪列表中移除 */
21      //OS_RdyListRemove(OSTCBCurPtr);
22      OS_PrioRemove(OSTCBCurPtr->Prio);                       (3)
23
24      /* 退出临界区 */
25      OS_CRITICAL_EXIT();                                     (4)
26
27      /* 任务调度 */
28      OSSched();
29  }
```

代码清单 11-9（1）：定义一个局部变量，用来存 CPU 关中断前的中断状态，因为接下来将任务从就绪列表移除这段代码属于临界短代码，需要关中断。

代码清单 11-9（2）：进入临界段

代码清单 11-9（3）：将任务从就绪列表移除，这里只需将任务在优先级表中对应的位清除即可，暂时不需要把任务 TCB 从 OSRdyList[]中移除，因为接下来 OSTimeTick()函数还是通过扫描 OSRdyList[]来判断任务的延时时间是否到期。当我们加入了时基列表之后，当任务调用 OSTimeDly()函数进行延时，就可以把任务的 TCB 从就绪列表删除，然后把任务 TCB 插入时基列表，OSTimeTick()函数判断任务的延时是否到期只需通过扫描时基列表即可，时基列表在下一个章节实现。所以这里暂时不能把 TCB 从就绪列表中删除，只是将任务优先级在优先级表中对应的位清除来达到任务不处于就绪态的目的。

代码清单 11-9（4）：退出临界段。

11.9 修改 OSSched()函数

任务调度函数 OSSched()不再是之前的两个任务轮流切换，需要根据优先级来调度，具体修改部分见代码清单 11-10 的加粗部分，被迭代的代码已经通过条件编译屏蔽。

代码清单 11-10 OSSched()函数

```
1  void OSSched(void)
2  {
3  #if 0
4      /* 如果当前任务是空闲任务，那么就去尝试执行任务 1 或者任务 2，
5         看看他们的延时时间是否结束，如果任务的延时时间均没有到期，
6         那就返回继续执行空闲任务 */
7      if ( OSTCBCurPtr == &OSIdleTaskTCB ) {
8          if (OSRdyList[0].HeadPtr->TaskDelayTicks == 0) {
9              OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
10         } else if (OSRdyList[1].HeadPtr->TaskDelayTicks == 0) {
11             OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
12         } else {
13             return;    /* 任务延时均没有到期则返回，继续执行空闲任务 */
14         }
15     } else {
16         /*如果是 task1 或者 task2 的话，检查下另外一个任务，
17         如果另外的任务不在延时中，就切换到该任务，
18         否则，判断下当前任务是否应该进入延时状态，
19         如果是的话，就切换到空闲任务。否则就不进行任何切换 */
20         if (OSTCBCurPtr == OSRdyList[0].HeadPtr) {
21             if (OSRdyList[1].HeadPtr->TaskDelayTicks == 0) {
```

```
22                  OSTCBHighRdyPtr = OSRdyList[1].HeadPtr;
23              } else if (OSTCBCurPtr->TaskDelayTicks != 0) {
24                  OSTCBHighRdyPtr = &OSIdleTaskTCB;
25              } else {
26                  /* 返回，不进行切换，因为两个任务都处于延时中 */
27                  return;
28              }
29          } else if (OSTCBCurPtr == OSRdyList[1].HeadPtr) {
30              if (OSRdyList[0].HeadPtr->TaskDelayTicks == 0) {
31                  OSTCBHighRdyPtr = OSRdyList[0].HeadPtr;
32              } else if (OSTCBCurPtr->TaskDelayTicks != 0) {
33                  OSTCBHighRdyPtr = &OSIdleTaskTCB;
34              } else {
35                  /* 返回，不进行切换，因为两个任务都处于延时中 */
36                  return;
37              }
38          }
39      }
40
41      /* 任务切换 */
42      OS_TASK_SW();
43  #endif
44
45      CPU_SR_ALLOC();                                                    (1)
46
47      /* 进入临界区 */
48      OS_CRITICAL_ENTER();                                               (2)
49
50      /* 查找最高优先级的任务 */                                          (3)
51      OSPrioHighRdy   = OS_PrioGetHighest();
52      OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
53
54      /* 如果最高优先级的任务是当前任务则直接返回，不进行任务切换 */        (4)
55      if (OSTCBHighRdyPtr == OSTCBCurPtr) {
56          /* 退出临界区 */
57          OS_CRITICAL_EXIT();
58
59          return;
60      }
61      /* 退出临界区 */
62      OS_CRITICAL_EXIT();                                                (5)
63
64      /* 任务切换 */
65      OS_TASK_SW();                                                      (6)
66  }
```

代码清单 11-10（1）：定义一个局部变量，用来存 CPU 关中断前的中断状态，因为接下来查找最高优先级这段代码属于临界短代码，需要关中断。

代码清单 11-10（2）：进入临界段。

代码清单 11-10（3）：查找最高优先级任务。

代码清单 11-10（4）：判断最高优先级任务是不是当前任务，如果是则直接返回，否则将继续往下执行，最后执行任务切换。

代码清单 11-10（5）：退出临界段。

代码清单 11-10（6）：任务切换。

11.10 修改 OSTimeTick()函数

OSTimeTick()函数在 SysTick 中断服务函数中被调用，是一个周期函数，具体用于扫描就绪列表 OSRdyList[]，判断任务的延时时间是否到期，如果到期则将任务在优先级表中对应的位置位，修改部分的代码见代码清单 11-11 的加粗部分，被迭代的代码则通过条件编译屏蔽。

代码清单 11-11 OSTimeTick()函数

```
1  void  OSTimeTick (void)
2  {
3      unsigned int i;
4      CPU_SR_ALLOC();                                                    (1)
5
6      /* 进入临界区 */
7      OS_CRITICAL_ENTER();                                               (2)
8
9      /* 扫描就绪列表中所有任务的 TaskDelayTicks，如果不为 0，则减 1 */
10 #if 0
11     for (i=0; i<OS_CFG_PRIO_MAX; i++) {
12         if (OSRdyList[i].HeadPtr->TaskDelayTicks > 0) {
13             OSRdyList[i].HeadPtr->TaskDelayTicks --;
14         }
15     }
16 #endif
17
18     for (i=0; i<OS_CFG_PRIO_MAX; i++) {                                (3)
19         if (OSRdyList[i].HeadPtr->TaskDelayTicks > 0) {
20             OSRdyList[i].HeadPtr->TaskDelayTicks --;
21             if (OSRdyList[i].HeadPtr->TaskDelayTicks == 0) {
22                 /* 为 0 则表示延时时间到，让任务就绪 */
23                 //OS_RdyListInsert (OSRdyList[i].HeadPtr);
24                 OS_PrioInsert(i);
25             }
26         }
27     }
28
29     /* 退出临界区 */
30     OS_CRITICAL_EXIT();                                                (4)
31
32     /* 任务调度 */
33     OSSched();
34 }
```

代码清单 11-11（1）：定义一个局部变量，用来存 CPU 关中断前的中断状态，因为接下来扫描就绪列表 OSRdyList[]这段代码属于临界短代码，需要关中断。

代码清单 11-11（2）：进入临界段。

代码清单 11-11（3）：扫描就绪列表 OSRdyList[]，判断任务的延时时间是否到期，如果到期则将任务在优先级表中对应的位置位。

代码清单 11-11（4）：退出临界段。

11.11 main 函数

main 函数具体见代码清单 11-12，修改部分代码已经加粗显示。

代码清单 11-12 main()函数

```
1  /*
```

```
2  *********************************************************************
3  *                                 全局变量
4  *********************************************************************
5  */
6
7  uint32_t flag1;
8  uint32_t flag2;
9  uint32_t flag3;
10
11 /*
12 *********************************************************************
13 *                          TCB & STACK & 任务声明
14 *********************************************************************
15 */
16 #define  TASK1_STK_SIZE        128
17 #define  TASK2_STK_SIZE        128
18 #define  TASK3_STK_SIZE        128
19
20
21 static   OS_TCB     Task1TCB;
22 static   OS_TCB     Task2TCB;
23 static   OS_TCB     Task3TCB;
24
25
26 static   CPU_STK    Task1Stk[TASK1_STK_SIZE];
27 static   CPU_STK    Task2Stk[TASK2_STK_SIZE];
28 static   CPU_STK    Task3Stk[TASK2_STK_SIZE];
29
30
31 void     Task1( void *p_arg );
32 void     Task2( void *p_arg );
33 void     Task3( void *p_arg );
34
35
36 /*
37 *********************************************************************
38 *                                 函数声明
39 *********************************************************************
40 */
41 void delay(uint32_t count);
42
43 /*
44 *********************************************************************
45 *                                 main 函数
46 *********************************************************************
47 */
48 /*
49 * 注意事项：1、该工程使用软件仿真，debug 需选择 Ude Simulator
50 *           2、在 Target 选项卡里面把晶振 Xtal(Mhz)的值改为 25，默认是 12，
51 *              改成 25 是为了跟 system_ARMCM3.c 中定义的__SYSTEM_CLOCK 相同，
52 *              确保仿真的时候时钟一致
53 */
54 int main(void)
55 {
56     OS_ERR err;
57
58
59     /* CPU 初始化：1、初始化时间戳 */
60     CPU_Init();
61
62     /* 关闭中断 */
63     CPU_IntDis();
64
65     /* 配置 SysTick 10ms 中断一次 */
66     OS_CPU_SysTickInit (10);
```

```
67
68      /* 初始化相关的全局变量 */
69      OSInit(&err);                                               (1)
70
71      /* 创建任务 */
72      OSTaskCreate( (OS_TCB*)&Task1TCB,
73                    (OS_TASK_PTR )Task1,
74                    (void *)0,
75                    (OS_PRIO)1,                                   (2)
76                    (CPU_STK*)&Task1Stk[0],
77                    (CPU_STK_SIZE)  TASK1_STK_SIZE,
78                    (OS_ERR *)&err );
79
80      OSTaskCreate( (OS_TCB*)&Task2TCB,
81                    (OS_TASK_PTR )Task2,
82                    (void *)0,
83                    (OS_PRIO)2,                                   (3)
84                    (CPU_STK*)&Task2Stk[0],
85                    (CPU_STK_SIZE)  TASK2_STK_SIZE,
86                    (OS_ERR *)&err );
87
88      OSTaskCreate( (OS_TCB*)&Task3TCB,
89                    (OS_TASK_PTR )Task3,
90                    (void *)0,
91                    (OS_PRIO)3,                                   (4)
92                    (CPU_STK*)&Task3Stk[0],
93                    (CPU_STK_SIZE)  TASK3_STK_SIZE,
94                    (OS_ERR *)&err );
95  #if 0
96      /* 将任务加入到就绪列表 */                                   (5)
97      OSRdyList[0].HeadPtr = &Task1TCB;
98      OSRdyList[1].HeadPtr = &Task2TCB;
99  #endif
100
101     /* 启动 OS，将不再返回 */
102     OSStart(&err);
103 }
104
105 /*
106 *********************************************************************
107 *                                函数实现
108 *********************************************************************
109 */
110 /* 软件延时 */
111 void delay (uint32_t count)
112 {
113     for (; count!=0; count--);
114 }
115
116
117
118 void Task1( void *p_arg )
119 {
120     for ( ;; ) {
121         flag1 = 1;
122         OSTimeDly(2);
123         flag1 = 0;
124         OSTimeDly(2);
125     }
126 }
127
128 void Task2( void *p_arg )
129 {
130     for ( ;; ) {
131         flag2 = 1;
132         OSTimeDly(2);
```

```
133          flag2 = 0;
134          OSTimeDly(2);
135      }
136  }
137
138  void Task3( void *p_arg )
139  {
140      for ( ;; ) {
141          flag3 = 1;
142          OSTimeDly(2);
143          flag3 = 0;
144          OSTimeDly(2);
145      }
146  }
```

代码清单 11-12（1）：加入了优先级相关的全局变量 OSPrioCur 和 OSPrioHighRdy 的初始化。

代码清单 11-12（2）、（3）和（4）：为每个任务分配了优先级，任务 1 的优先级为 1，任务 2 的优先级为 2，任务 3 的优先级为 3。

代码清单 11-12（5）：将任务插入到就绪列表这部分功能由 OSTaskCreate()实现，这里通过条件编译屏蔽掉。

11.12 实验现象

进入软件调试，全速运行程序，从逻辑分析仪中可以看到三个任务的波形是完全同步，就好像 CPU 在同时干三件事情，具体仿真的波形图见图 11-1。任务开始的启动过程具体见图 11-2，这个启动过程要认真的理解下。

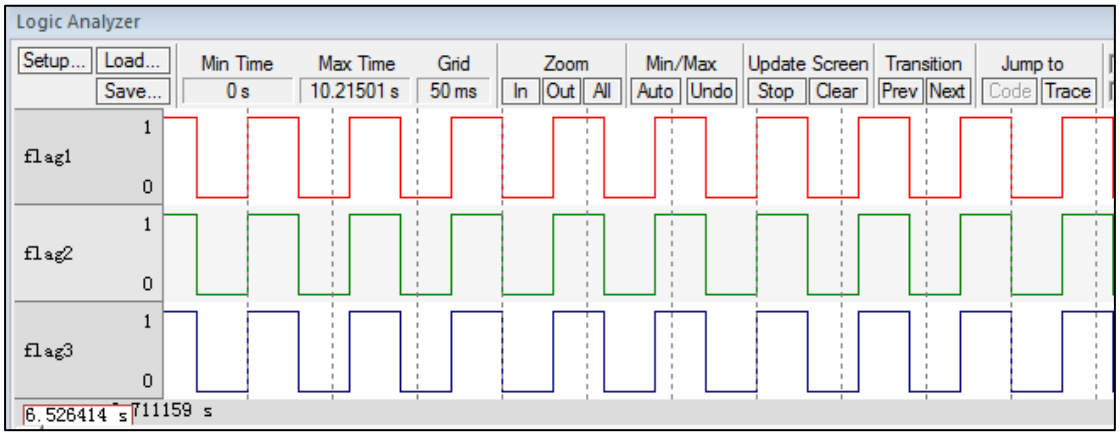


图 11-1 实验现象（宏观）

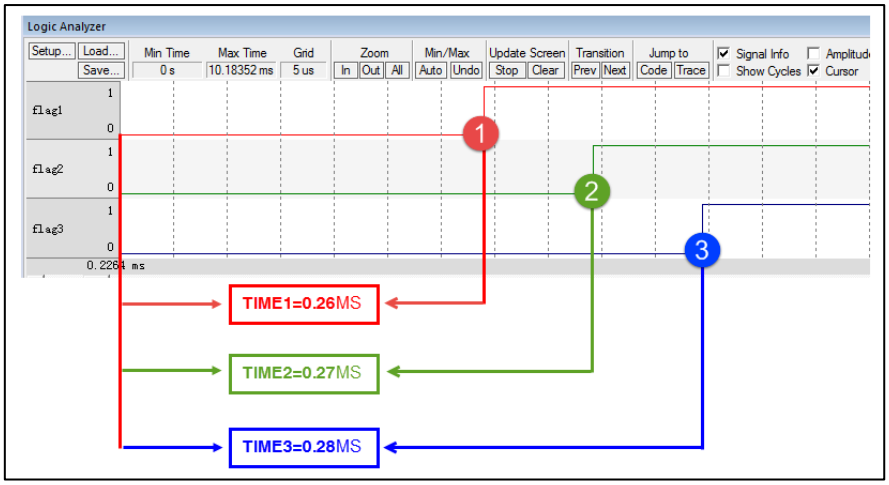


图 11-2 任务的启动过程（微观）

图 11-2 是任务 1、2 和 3 刚开始启动时的软件仿真波形图，系统从启动到任务 1 开始运行前花的时间为 TIME1，等于 0.26MS。任务 1 开始运行，然后调用 OSTimeDly(1)进入延时，随后进行任务切换，切换到任务 2 开始运行，从任务 1 切换到任务 2 花费的时间等于 TIME2-TIME1，等于 0.01MS。任务 2 开始运行，然后调用 OSTimeDly(1)进入延时，随后进行任务切换，切换到任务 3 开始运行，从任务 2 切换到任务 3 花费的时间等于 TIME3-TIME1，等于 0.01MS。任务 3 开始运行，然后调用 OSTimeDly(1)进入延时，随后进行任务切换，这个时候我们创建的 3 个任务都处于延时状态，那么系统就切换到空闲任务，在三个任务延时未到期之前，系统一直都是在运行空闲任务。当第一个 SysTick 中断产生，中断服务函数会调用 OSTimeTick() 函数扫描每个任务的延时是否到期，因为是延时 1 个 SysTick 周期，所以第一个 SysTick 中断产生就意味着延时都到期，任务 1、2 和 3 依次进入就绪态，再次回到任务本身接着运行，将自身的 Flag 清 0，然后任务 1、2 和 3 又依次调用 OSTimeDly(1)进入延时状态，直到下一个 SysTick 中断产生前，系统都处在空闲任务中，一直这样循环下去。

但是，有些同学肯定就会问图 11-1 中任务 1、2 和 3 的波形图是同步的，而图 11-2 中任务的波形就不同步，有先后顺序？答案是图 11-2 是将两个任务切换花费的时间 0.01ms 进行放大后观察的波形，就好像我们用放大镜看微小的东西一样，如果不用放大镜，在宏观层面观察就是图 11-1 的实验现象。

第12章　实现时基列表

从本章开始，我们在 OS 中加入时基列表，时基列表是跟时间相关的，处于延时的任务和等待事件有超时限制的任务都会从就绪列表中移除，然后插入到时基列表。时基列表在 OSTimeTick 中更新，如果任务的延时时间结束或者超时到期，就会让任务就绪，从时基列表移除，插入到就绪列表。到目前为止，我们在 OS 中只实现了两个列表，一个是就绪列表，一个是本章将要实现的时基列表，在本章之前，任务要么在就绪列表，要么在时基列表。

12.1 实现时基列表

12.1.1 定义时基列表变量

时基列表在代码层面上由全局数组 OSCfg_TickWheel[]和全局变量 OSTickCtr 构成，一个空的时基列表示意图见图 12-1，时基列表的代码实现具体见代码清单 12-1。

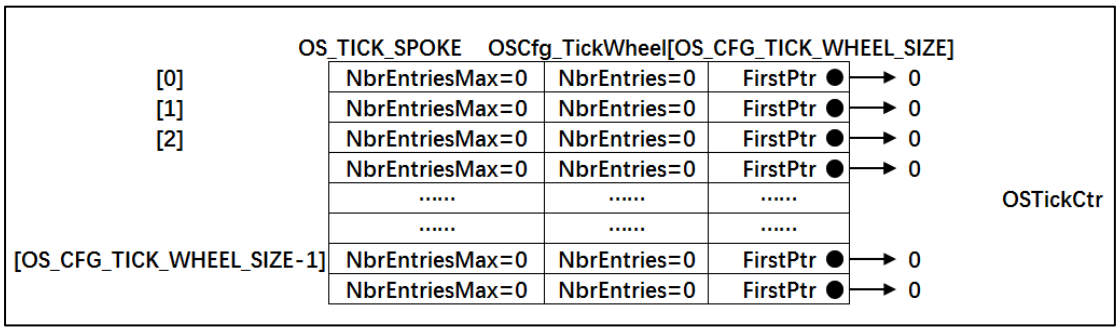


图 12-1 空的时基列表

代码清单 12-1 时基列表定义

```
1  /* 时基列表大小，在 os_cfg_app.h 定义 */
2  #define  OS_CFG_TICK_WHEEL_SIZE            17u
3
4  /* 在 os_cfg_app.c 定义 */
5  /* 时基列表 */
6       (1)                                   (2)
7  OS_TICK_SPOKE  OSCfg_TickWheel[OS_CFG_TICK_WHEEL_SIZE];
8  /* 时基列表大小 */
9  OS_OBJ_QTY const OSCfg_TickWheelSize = (OS_OBJ_QTY  )OS_CFG_TICK_WHEEL_SIZE;
10
11
12 /* 在 os.h 中声明 */
13 /* 时基列表 */
14 extern  OS_TICK_SPOKE  OSCfg_TickWheel[];
15 /* 时基列表大小 */
16 extern  OS_OBJ_QTY    const OSCfg_TickWheelSize;
17
18
19 /* Tick 计数器，在 os.h 中定义 */
20 OS_EXT            OS_TICK                 OSTickCtr;        (3)
```

代码清单 12-1 （1）：OS_TICK_SPOKE 为时基列表数组 OSCfg_TickWheel[]的数据类型，在 os.h 文件定义，具体见代码清单 12-2。

代码清单 12-2 OS_TICK_SPOKE 定义

```
1 typedef  struct  os_tick_spoke       OS_TICK_SPOKE;       (1)
2
3 struct  os_tick_spoke {
4     OS_TCB              *FirstPtr;                        (2)
5     OS_OBJ_QTY           NbrEntries;                      (3)
6     OS_OBJ_QTY           NbrEntriesMax;                   (4)
7 };
```

代码清单 12-2（1）：在 uC/OS-III 中，内核对象的数据类型都会用大写字母重新定义。

代码清单 12-2（2）：时基列表 OSCfg_TickWheel[]的每个成员都包含一条单向链表，被插入到该条链表的 TCB 会按照延时时间做升序排列。FirstPtr 用于指向这条单向链表的第一个节点。

代码清单 12-2（3）：时基列表 OSCfg_TickWheel[]的每个成员都包含一条单向链表，NbrEntries 表示该条单向链表当前有多少个节点。

代码清单 12-2（4）：时基列表 OSCfg_TickWheel[]的每个成员都包含一条单向链表，NbrEntriesMax 记录该条单向链表最多的时候有多少个节点，在增加节点的时候会刷新，在删除节点的时候不刷新。

代码清单 12-1（2）：OS_CFG_TICK_WHEEL_SIZE 是一个宏，在 os_cfg_app.h 中定义，用于控制时基列表的大小。OS_CFG_TICK_WHEEL_SIZE 的推荐值为 任务数/4，不推荐使用偶数，如果算出来是偶数，则加 1 变成质数，实际上质数是一个很好的选择。

代码清单 12-1（3）：OSTickCtr 为 SysTick 周期计数器，记录系统启动到现在或者从上一次复位到现在经过了多少个 SysTick 周期。

12.1.2 修改任务控制块 TCB

时基列表 OSCfg_TickWheel[]的每个成员都包含一条单向链表，被插入到该条链表的 TCB 会按照延时时间做升序排列，为了 TCB 能按照延时时间从小到大串接在一起，需要在 TCB 中加入几个成员，具体见代码清单 12-3 的加粗部分。

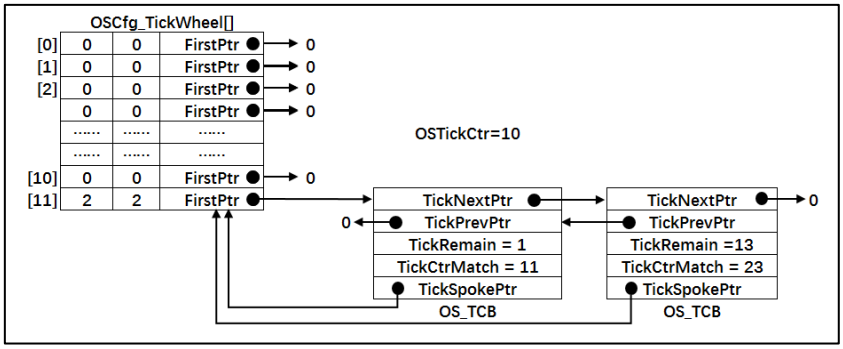


图 12-2 时基列表中有两个 TCB

代码清单 12-3 在 TCB 中加入时基列表相关字段

```
1  struct os_tcb {
2      CPU_STK          *StkPtr;
3      CPU_STK_SIZE     StkSize;
4
5      /* 任务延时周期个数 */
6      OS_TICK          TaskDelayTicks;
7
8      /* 任务优先级 */
9      OS_PRIO          Prio;
10
11     /* 就绪列表双向链表的下一个指针 */
12     OS_TCB           *NextPtr;
13     /* 就绪列表双向链表的前一个指针 */
14     OS_TCB           *PrevPtr;
15
16     /*时基列表相关字段*/
17     OS_TCB           *TickNextPtr;          (1)
18     OS_TCB           *TickPrevPtr;          (2)
19     OS_TICK_SPOKE    *TickSpokePtr;         (5)
20
21     OS_TICK          TickCtrMatch;          (4)
22     OS_TICK          TickRemain;            (3)
23 };
```

代码清单 12-3 加粗部分的字段可以配合图 12-2 一起理解，这样会比较容易。图 12-2 是在时基列表 OSCfg_TickWheel[]索引 11 这条链表里面插入了两个 TCB，一个需要延时 1 个时钟周期，另外一个需要延时 13 个时钟周期。

代码清单 12-3（1）：TickNextPtr 用于指向链表中的下一个 TCB 节点。

代码清单 12-3（2）：TickPrevPtr 用于指向链表中的上一个 TCB 节点。

代码清单 12-3（3）：TickRemain 用于设置任务还需要等待多少个时钟周期，每到来一个时钟周期，该值会递减。

代码清单 12-3（4）：TickCtrMatch 的值等于时基计数器 OSTickCtr 的值加上 TickRemain 的值，当 TickCtrMatch 的值等于 OSTickCtr 的值的时候，表示等待到期，TCB 会从链表中删除。

代码清单 12-3（5）：每个被插入到链表的 TCB 都包含一个字段 TickSpokePtr，用于回指到链表的根部。

12.1.3 实现时基列表相关函数

时基列表相关函数在 os_tick.c 实现，在 os.h 中声明。如果 os_tick.c 文件是第一次使用，需要自行在文件夹 uCOS-III\Source 中新建并添加到工程的 uC/OS-III Source 组。

1. OS_TickListInit()函数

OS_TickListInit()函数用于初始化时基列表，即将全局变量 OSCfg_TickWheel[]的数据域全部初始化为 0，一个初始化为 0 的的时基列表见图 12-3。

代码清单 12-4 OS_TickListInit()函数

```
1 /* 初始化时基列表的数据域 */
2 void  OS_TickListInit (void)
3 {
```

```
    OS_TICK_SPOKE_IX   i;
    OS_TICK_SPOKE     *p_spoke;

    for (i = 0u; i < OSCfg_TickWheelSize; i++) {
        p_spoke                = (OS_TICK_SPOKE *)&OSCfg_TickWheel[i];
        p_spoke->FirstPtr      = (OS_TCB        *)0;
        p_spoke->NbrEntries    = (OS_OBJ_QTY     )0u;
        p_spoke->NbrEntriesMax = (OS_OBJ_QTY     )0u;
    }
}
```

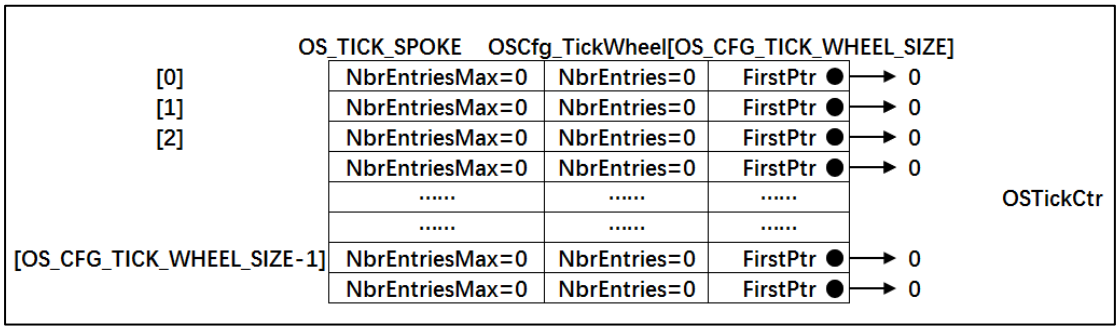


图 12-3 时基列表的数据域全部被初始化为 0，即为空

2. OS_TickListInsert()函数

OS_TickListInsert()函数用于往时基列表中插入一个任务 TCB，具体实现见代码清单 12-5。代码清单 12-5 可配和图 12-4 一起阅读，这样理解起来会容易很多。

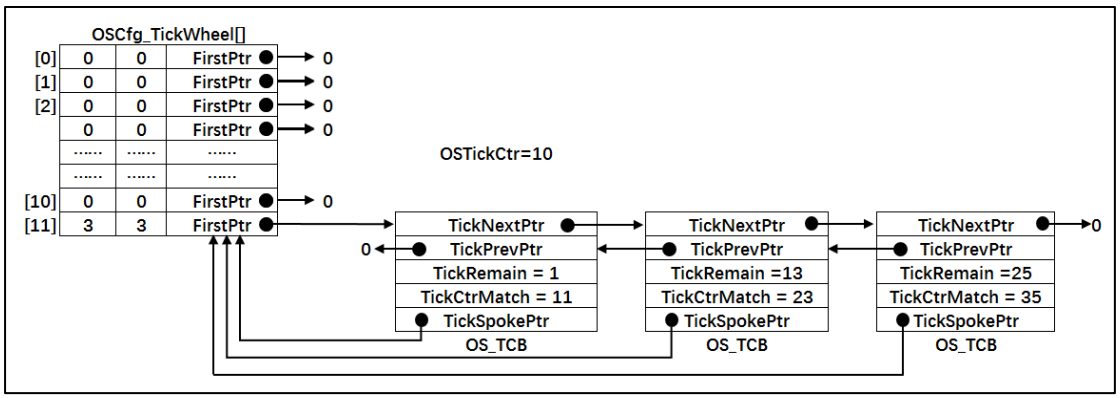


图 12-4 时基列表中有三个 TCB

代码清单 12-5 OS_TickListInsert()函数

```
/* 将一个任务插入到时基列表，根据延时时间的大小升序排列 */
void  OS_TickListInsert (OS_TCB *p_tcb,OS_TICK time)
{
    OS_TICK_SPOKE_IX   spoke;
    OS_TICK_SPOKE     *p_spoke;
    OS_TCB            *p_tcb0;
    OS_TCB            *p_tcb1;

    p_tcb->TickCtrMatch = OSTickCtr + time;                          (1)
    p_tcb->TickRemain   = time;                                      (2)

    spoke   = (OS_TICK_SPOKE_IX)(p_tcb->TickCtrMatch % OSCfg_TickWheelSize);  (3)
```

```
13      p_spoke = &OSCfg_TickWheel[spoke];                                  (4)
14
15      /* 插入到 OSCfg_TickWheel[spoke] 的第一个节点 */
16      if (p_spoke->NbrEntries == (OS_OBJ_QTY)0u) {                        (5)
17          p_tcb->TickNextPtr    = (OS_TCB   *)0;
18          p_tcb->TickPrevPtr    = (OS_TCB   *)0;
19          p_spoke->FirstPtr     =  p_tcb;
20          p_spoke->NbrEntries   = (OS_OBJ_QTY)1u;
21      }
22      /* 如果插入的不是第一个节点，则按照 TickRemain 大小升序排列 */
23      else {                                                              (6)
24          /* 获取第一个节点指针 */
25          p_tcb1 = p_spoke->FirstPtr;
26          while (p_tcb1 != (OS_TCB *)0) {
27              /* 计算比较节点的剩余时间 */
28              p_tcb1->TickRemain = p_tcb1->TickCtrMatch - OSTickCtr;
29
30              /* 插入到比较节点的后面 */
31              if (p_tcb->TickRemain > p_tcb1->TickRemain) {
32                  if (p_tcb1->TickNextPtr != (OS_TCB *)0) {
33                      /* 寻找下一个比较节点 */
34                      p_tcb1 =  p_tcb1->TickNextPtr;
35                  } else {  /* 在最后一个节点插入 */
36                      p_tcb->TickNextPtr   = (OS_TCB *)0;
37                      p_tcb->TickPrevPtr   =  p_tcb1;
38                      p_tcb1->TickNextPtr  =  p_tcb;
39                      p_tcb1               = (OS_TCB *)0;        (7)
40                  }
41              }
42              /* 插入到比较节点的前面 */
43              else {
44                  /* 在第一个节点插入 */
45                  if (p_tcb1->TickPrevPtr == (OS_TCB *)0) {
46                      p_tcb->TickPrevPtr   = (OS_TCB *)0;
47                      p_tcb->TickNextPtr   =  p_tcb1;
48                      p_tcb1->TickPrevPtr  =  p_tcb;
49                      p_spoke->FirstPtr    =  p_tcb;
50                  } else {
51                      /* 插入到两个节点之间 */
52                      p_tcb0               =  p_tcb1->TickPrevPtr;
53                      p_tcb->TickPrevPtr   =  p_tcb0;
54                      p_tcb->TickNextPtr   =  p_tcb1;
55                      p_tcb0->TickNextPtr  =  p_tcb;
56                      p_tcb1->TickPrevPtr  =  p_tcb;
57                  }
58                  /* 跳出 while 循环 */
59                  p_tcb1 = (OS_TCB *)0;                             (8)
60              }
61          }
62
63          /* 节点成功插入 */
64          p_spoke->NbrEntries++;                                    (9)
65      }
66
67      /* 刷新 NbrEntriesMax 的值 */
68      if (p_spoke->NbrEntriesMax < p_spoke->NbrEntries) {           (10)
69          p_spoke->NbrEntriesMax = p_spoke->NbrEntries;
70      }
71
72      /* 任务 TCB 中的 TickSpokePtr 回指根节点 */
73      p_tcb->TickSpokePtr = p_spoke;                                (11)
74  }
```

代码清单 12-5（1）：TickCtrMatch 的值等于当前时基计数器的值 OSTickCtr 加上任务要延时的时间 time，time 由函数形参传进来。OSTickCtr 是一个全局变量，记录的是系统自启动以来或者自上次复位以来经过了多少个 SysTick 周期。OSTickCtr 的值每经过一个 SysTick 周期其值就加一，当 TickCtrMatch 的值与其相等时，就表示任务等待时间到期。

代码清单 12-5（2）：将任务需要延时的时间 time 保存到 TCB 的 TickRemain，它表示任务还需要延时多少个 SysTick 周期，每到来一个 SysTick 周期，TickRemain 会减一。

代码清单 12-5（3）：由任务的 TickCtrMatch 对时基列表的大小 OSCfg_TickWheelSize 进行求余操作，得出的值 spoke 作为 时基列表 OSCfg_TickWheel[]的索引。只要是任务的 TickCtrMatch 对 OSCfg_TickWheelSize 求余后得到的值 spoke 相等，那么任务的 TCB 就会被插入到 OSCfg_TickWheel[spoke]下的单向链表中，节点按照任务的 TickCtrMatch 值做升序排列。举例：在图 12-4 中，时基列表 OSCfg_TickWheel[]的大小 OSCfg_TickWheelSize 等于 12，当前时基计数器 OSTickCtr 的值为 10，有三个任务分别需要延时 TickTemain=1、TickTemain=23 和 TickTemain=25 个时钟周期，三个任务的 TickRemain 加上 OSTickCtr 可分别得出它们的 TickCtrMatch 等于 11、23 和 35，这三个任务的 TickCtrMatch 对 OSCfg_TickWheelSize 求余操作后的值 spoke 都等于 11，所以这三个任务的 TCB 会被插入到 OSCfg_TickWheel[11]下的同一条链表，节点顺序根据 TickCtrMatch 的值做升序排列。

代码清单 12-5（4）：根据刚刚算出的索引值 spoke，获取到该索引值下的成员的地址，也叫根指针，因为该索引下对应的成员 OSCfg_TickWheel[spoke]会维持一条单向的链表。

代码清单 12-5（5）：将 TCB 插入到链表中分两种情况，第一是当前链表是空的，插入的节点将成为第一个节点，这个处理非常简单；第二是当前链表已经有节点。

代码清单 12-5（6）：当前的链表中已经有节点，插入的时候则根据 TickCtrMatch 的值做升序排列，插入的时候分三种情况，第一是在最后一个节点之间插入，第二是在第一个节点插入，第三是在两个节点之间插入。

代码清单 12-5（7）（8）：节点成功插入 p_tcb1 指针，跳出 while 循环

代码清单 12-5（9）：节点成功插入，记录当前链表节点个数的计数器 NbrEntries 加一。

代码清单 12-5（10）：刷新 NbrEntriesMax 的值, NbrEntriesMax 用于记录当前链表曾经最多有多少个节点，只有在增加节点的时候才刷新，在删除节点的时候是不刷新的。

代码清单 12-5（11）：任务 TCB 被成功插入到链表，TCB 中的 TickSpokePtr 回指所在链表的根指针。

3. OS_TickListRemove()函数

OS_TickListRemove()用于从时基列表删除一个指定的 TCB 节点，具体实现见。代码清单 12-6

代码清单 12-6 OS_TickListRemove()函数

```
1 /* 从时基列表中移除一个任务 */
2 void  OS_TickListRemove (OS_TCB  *p_tcb)
3 {
4     OS_TICK_SPOKE  *p_spoke;
5     OS_TCB         *p_tcb1;
6     OS_TCB         *p_tcb2;
```

```
 7
 8      /* 获取任务 TCB 所在链表的根指针 */
 9      p_spoke = p_tcb->TickSpokePtr;                                  (1)
10
11      /* 确保任务在链表中 */
12      if (p_spoke != (OS_TICK_SPOKE *)0) {
13          /* 将剩余时间清 0 */
14          p_tcb->TickRemain = (OS_TICK)0u;
15
16          /* 要移除的刚好是第一个节点 */
17          if (p_spoke->FirstPtr == p_tcb) {                           (2)
18              /* 更新第一个节点，原来的第一个节点需要被移除 */
19              p_tcb1             = (OS_TCB *)p_tcb->TickNextPtr;
20              p_spoke->FirstPtr = p_tcb1;
21              if (p_tcb1 != (OS_TCB *)0) {
22                  p_tcb1->TickPrevPtr = (OS_TCB *)0;
23              }
24          }
25          /* 要移除的不是第一个节点 */                                  (3)
26          else {
27              /* 保存要移除的节点的前后节点的指针 */
28              p_tcb1              = p_tcb->TickPrevPtr;
29              p_tcb2              = p_tcb->TickNextPtr;
30
31              /* 节点移除，将节点前后的两个节点连接在一起 */
32              p_tcb1->TickNextPtr = p_tcb2;
33              if (p_tcb2 != (OS_TCB *)0) {
34                  p_tcb2->TickPrevPtr = p_tcb1;
35              }
36          }
37
38          /* 复位任务 TCB 中时基列表相关的字段成员 */                   (4)
39          p_tcb->TickNextPtr  = (OS_TCB        *)0;
40          p_tcb->TickPrevPtr  = (OS_TCB        *)0;
41          p_tcb->TickSpokePtr = (OS_TICK_SPOKE *)0;
42          p_tcb->TickCtrMatch = (OS_TICK        )0u;
43
44          /* 节点减 1 */
45          p_spoke->NbrEntries--;                                      (5)
46      }
47  }
```

代码清单 12-6（1）：获取任务 TCB 所在链表的根指针。

代码清单 12-6（2）：要删除的节点是链表的第一个节点，这个操作很好处理，只需更新下第一个节点即可。

代码清单 12-6（3）：要删除的节点不是链表的第一个节点，则先保存要删除的节点的前后节点，然后把这前后两个节点相连即可。

代码清单 12-6（4）：复位任务 TCB 中时基列表相关的字段成员。

代码清单 12-6（5）：节点删除成功，链表中的节点计数器 NbrEntries 减一。

4. OS_TickListUpdate()函数

OS_TickListUpdate()在每个 SysTick 周期到来时在 OSTimeTick()被调用，用于更新时基计数器 OSTickCtr，扫描时基列表中的任务延时是否到期，具体实现见代码清单 12-7。

代码清单 12-7 OS_TickListUpdate()函数

```
1  void  OS_TickListUpdate (void)
```

```
2  {
3      OS_TICK_SPOKE_IX   spoke;
4      OS_TICK_SPOKE     *p_spoke;
5      OS_TCB            *p_tcb;
6      OS_TCB            *p_tcb_next;
7      CPU_BOOLEAN        done;
8
9      CPU_SR_ALLOC();
10
11     /* 进入临界段 */
12     OS_CRITICAL_ENTER();
13
14     /* 时基计数器++ */
15     OSTickCtr++;                                                          (1)
16
17     spoke    = (OS_TICK_SPOKE_IX)(OSTickCtr % OSCfg_TickWheelSize);       (2)
18     p_spoke  = &OSCfg_TickWheel[spoke];
19
20     p_tcb    = p_spoke->FirstPtr;
21     done     = DEF_FALSE;
22
23     while (done == DEF_FALSE) {
24         if (p_tcb != (OS_TCB *)0) {                                       (3)
25             p_tcb_next = p_tcb->TickNextPtr;
26
27             p_tcb->TickRemain = p_tcb->TickCtrMatch - OSTickCtr;          (4)
28
29             /* 节点延时时间到 */
30             if (OSTickCtr == p_tcb->TickCtrMatch) {                       (5)
31                 /* 让任务就绪 */
32                 OS_TaskRdy(p_tcb);
33             } else {                                                      (6)
34                 /* 如果第一个节点延时期未满，则退出 while 循环
35                    因为链表是根据升序排列的，第一个节点延时期未满，那后面的肯定未满
*/
36                 done = DEF_TRUE;
37             }
38
39             /* 如果第一个节点延时期满，则继续遍历链表，看看还有没有延时期满的任务
40                如果有，则让它就绪 */
41             p_tcb = p_tcb_next;                                           (7)
42         } else {
43             done  = DEF_TRUE;                                             (8)
44         }
45     }
46
47     /* 退出临界段 */
48     OS_CRITICAL_EXIT();
49 }
```

代码清单 12-7（1）：每到来一个 SysTick 时钟周期，时基计数器 OSTickCtr 都要加一操作。

代码清单 12-7（2）：计算要扫描的时基列表的索引，每次只扫描一条链表。时基列表里面有可能有多条链表，为啥只扫描其中一条链表就可以？因为任务在插入到时基列表的时候，插入的索引值 spoke_insert 是通过 TickCtrMatch 对 OSCfg_TickWheelSize 求余得出，现在需要扫描的索引值 spoke_update 是通过 OSTickCtr 对 OSCfg_TickWheelSize 求余得出，TickCtrMatch 的值等于 OSTickCt 加上 TickRemain，只有在经过 TickRemain 个时钟周期后，spoke_update 的值才有可能等于 spoke_insert。如果算出的 spoke_update 小于 spoke_insert，

且 OSCfg_TickWheel[spoke_update]下的链表的任务没有到期，那后面的肯定都没有到期，不用继续扫描。

举例，在图 12-5，时基列表 OSCfg_TickWheel[]的大小 OSCfg_TickWheelSize 等于 12，当前时基计数器 OSTickCtr 的值为 7，有三个任务分别需要延时 TickTemain=16、TickTemain=28 和 TickTemain=40 个时钟周期，三个任务的 TickRemain 加上 OSTickCtr 可分别得出它们的 TickCtrMatch 等于 23、35 和 47，这三个任务的 TickCtrMatch 对 OSCfg_TickWheelSize 求余操作后的值 spoke 都等于 11，所以这三个任务的 TCB 会被插入到 OSCfg_TickWheel[11]下的同一条链表，节点顺序根据 TickCtrMatch 的值做升序排列。当下一个 SysTick 时钟周期到来的时候，会调用 OS_TickListUpdate()函数，这时 OSTickCtr 加一操作后等于 8，对 OSCfg_TickWheelSize（等于 12）求余算得要扫描更新的索引值 spoke_update 等 8，则对 OSCfg_TickWheel[8]下面的链表进行扫描，从图 12-5 可以得知，8 这个索引下没有节点，则直接退出，刚刚插入的三个 TCB 是在 OSCfg_TickWheel[11]下的链表，根本不用扫描，因为时间只是刚刚过了 1 个时钟周期而已，远远没有达到他们需要的延时时间。

代码清单 12-7（3）：判断链表是否为空，为空则跳转到第（8）步骤。

代码清单 12-7（4）：链表不为空，递减第一个节点的 TickRemain。

代码清单 12-7（5）：判断第一个节点的延时时间是否到，如果到期，让任务就绪，即将任务从时基列表删除，插入就绪列表，这两步由函数 OS_TaskRdy()来完成，该函数在 os_core.c 中定义，具体实现见代码清单 12-8。

代码清单 12-8 OS_TaskRdy()函数

```
1 void  OS_TaskRdy (OS_TCB  *p_tcb)
2 {
3     /* 从时基列表删除 */
4     OS_TickListRemove(p_tcb);
5 
6     /* 插入就绪列表 */
7     OS_RdyListInsert(p_tcb);
8 }
```

代码清单 12-7（6）：如果第一个节点延时期未满，则退出 while 循环，因为链表是根据升序排列的，第一个节点延时期未满，那后面的肯定未满。

代码清单 12-7（7）：如果第一个节点延时到期，则继续判断下一个节点延时是否到期 。

代码清单 12-7（8）：链表为空，退出扫描，因为其它还没到期。

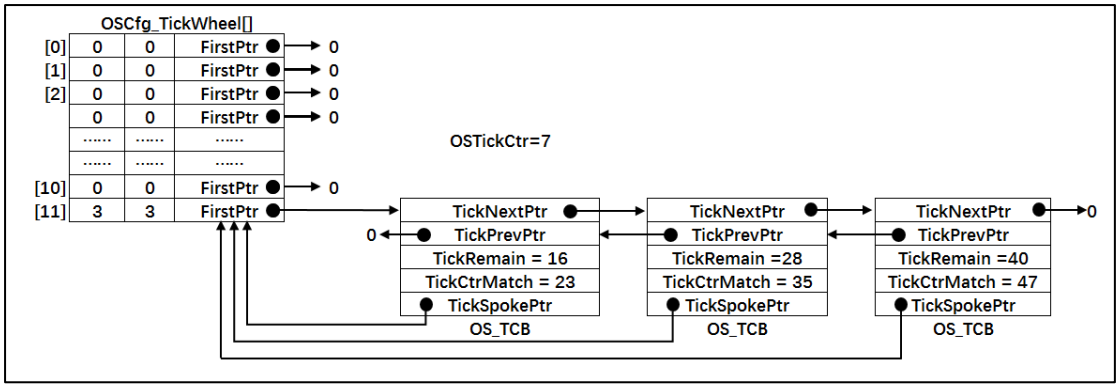


图 12-5 时基列表中有三个 TCB

12.2 修改 OSTimeDly()函数

加入时基列表之后，OSTimeDly()函数需要被修改，具体见代码清单 12-9 的加粗部分，被迭代的代码已经用条件编译屏蔽。

代码清单 12-9 OSTimeDly()函数

```
1  void  OSTimeDly(OS_TICK dly)
2  {
3      CPU_SR_ALLOC();
4
5      /* 进入临界区 */
6      OS_CRITICAL_ENTER();
7  #if 0
8      /* 设置延时时间 */
9      OSTCBCurPtr->TaskDelayTicks = dly;
10
11     /* 从就绪列表中移除 */
12     //OS_RdyListRemove(OSTCBCurPtr);
13     OS_PrioRemove(OSTCBCurPtr->Prio);
14 #endif
15
16     /* 插入到时基列表 */
17     OS_TickListInsert(OSTCBCurPtr, dly);
18
19     /* 从就绪列表移除 */
20     OS_RdyListRemove(OSTCBCurPtr);
21
22     /* 退出临界区 */
23     OS_CRITICAL_EXIT();
24
25     /* 任务调度 */
26     OSSched();
27 }
```

12.3 修改 OSTimeTick()函数

加入时基列表之后，OSTimeTick()函数需要被修改，具体见代码清单 12-10 的加粗部分，被迭代的代码已经用条件编译屏蔽。

代码清单 12-10 OSTimeTick()函数

```
1  void  OSTimeTick (void)
```

```
2  {
3  #if 0
4      unsigned int i;
5      CPU_SR_ALLOC();
6
7      /* 进入临界区 */
8      OS_CRITICAL_ENTER();
9
10     for (i=0; i<OS_CFG_PRIO_MAX; i++) {
11         if (OSRdyList[i].HeadPtr->TaskDelayTicks > 0) {
12             OSRdyList[i].HeadPtr->TaskDelayTicks --;
13             if (OSRdyList[i].HeadPtr->TaskDelayTicks == 0) {
14                 /* 为 0 则表示延时时间到，让任务就绪 */
15                 //OS_RdyListInsert (OSRdyList[i].HeadPtr);
16                 OS_PrioInsert(i);
17             }
18         }
19     }
20
21     /* 退出临界区 */
22     OS_CRITICAL_EXIT();
23
24 #endif
25
26     /* 更新时基列表 */
27     OS_TickListUpdate();
28
29     /* 任务调度 */
30     OSSched();
31 }
```

12.4 main 函数

main 函数同上一章一样。

12.5 实验现象

实验现象同上一章一样，实验现象虽然一样，但是任务在就是延时状态时，任务的 TCB 不再继续放在就绪列表，而是放在了时基列表中。

第13章　实现时间片

本章开始，我们让 OS 支持同一个优先级下可以有多个任务的功能，这些任务可以分配不同的时间片，当任务时间片用完的时候，任务会从链表的头部移动到尾部，让下一个任务共享时间片，以此循环。

13.1 实现时间片

13.1.1 修改任务 TCB

为了实现时间片功能，我们需要先在任务控制块 TCB 中添加两个时间片相关的变量，具体见代码清单 13-1 的加粗部分。

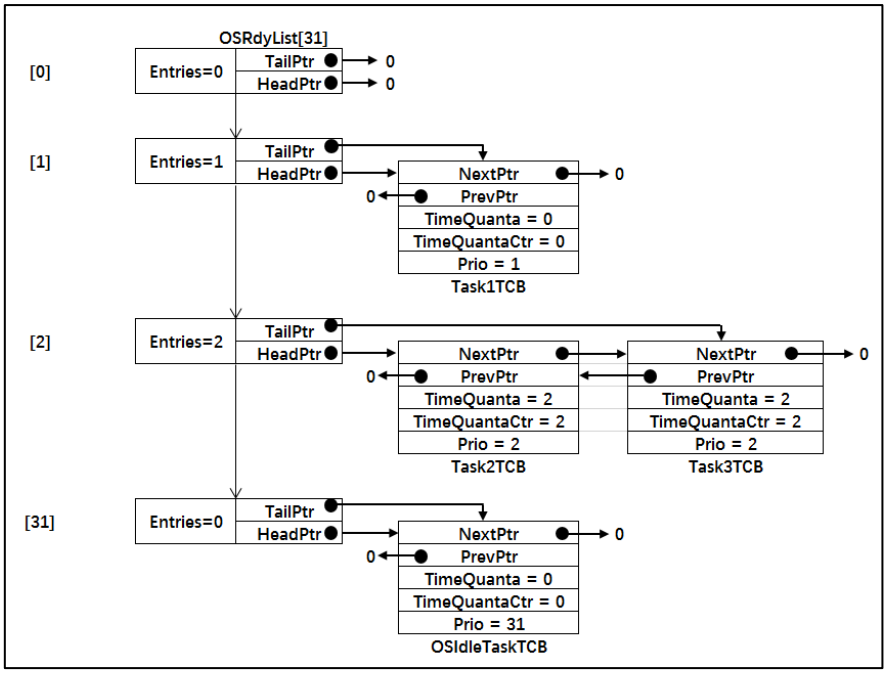
代码清单 13-1 在 TCB 中添加时间片相关的变量

```
1  struct os_tcb {
2      CPU_STK          *StkPtr;
3      CPU_STK_SIZE     StkSize;
4
5      /* 任务延时周期个数 */
6      OS_TICK          TaskDelayTicks;
7
8      /* 任务优先级 */
9      OS_PRIO          Prio;
10
11     /* 就绪列表双向链表的下一个指针 */
12     OS_TCB           *NextPtr;
13     /* 就绪列表双向链表的前一个指针 */
14     OS_TCB           *PrevPtr;
15
16     /*时基列表相关字段*/
17     OS_TCB           *TickNextPtr;
18     OS_TCB           *TickPrevPtr;
19     OS_TICK_SPOKE    *TickSpokePtr;
20
21     OS_TICK          TickCtrMatch;
22     OS_TICK          TickRemain;
23
24     /* 时间片相关字段 */
25     OS_TICK          TimeQuanta;          (1)
26     OS_TICK          TimeQuantaCtr;       (2)
27 };
```

代码清单 13-1（1）：TimeQuanta 表示任务需要多少个时间片，单位为系统时钟周期 Tick。

代码清单 13-1（2）：TimeQuantaCtr 表示任务还剩下多少个时间片，每到来一个系统时钟周期，TimeQuantaCtr 会减一，当 TimeQuantaCtr 等于零的时候，表示时间片用完，任

务的 TCB 会从就绪列表链表的头部移动到尾部，好让下一个任务共享时间片。



13.1.2 实现时间片调度函数

1. OS_SchedRoundRobin()函数

时间片调度函数 OS_SchedRoundRobin()在 os_core.c 中实现，在 OSTimeTick()调用，具体见代码清单 13-2。在阅读代码清单 13-2 的时候，可配图 13-1 一起理解，该图画的是在一个就绪链表中，有三个任务就绪，其中在优先级 2 下面有两个任务，均分配了两个时间片，其中任务 3 的时间片已用完，则位于链表的末尾，任务 2 的时间片还剩一个，则位于链表的头部。当下一个时钟周期到来的时候，任务 2 的时间片将耗完，相应的 TimeQuantaCtr 会递减为 0，任务 2 的 TCB 会被移动到链表的末尾，任务 3 则被成为链表的头部，然后重置任务 3 的时间片计数器 TimeQuantaCtr 的值为 2，重新享有时间片。

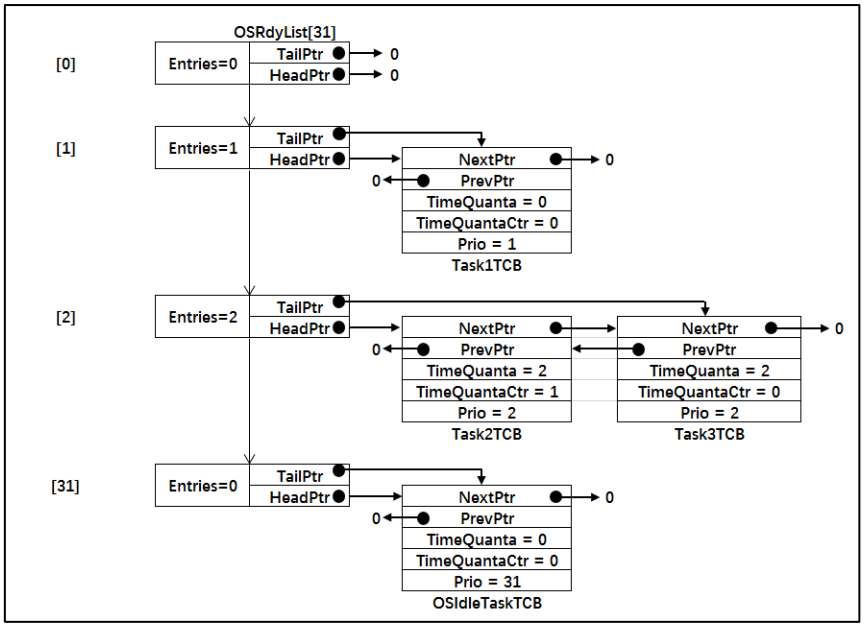


图 13-1 时间片调度函数讲解配套

代码清单 13-2 时间片调度函数

```
1  #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u                                    (1)
2  void OS_SchedRoundRobin(OS_RDY_LIST  *p_rdy_list)
3  {
4      OS_TCB   *p_tcb;
5      CPU_SR_ALLOC();
6
7      /*  进入临界段 */
8      CPU_CRITICAL_ENTER();
9
10     p_tcb = p_rdy_list->HeadPtr;                                        (2)
11
12     /* 如果 TCB 节点为空，则退出 */
13     if (p_tcb == (OS_TCB *)0) {                                         (3)
14         CPU_CRITICAL_EXIT();
15         return;
16     }
17
18     /* 如果是空闲任务，也退出 */
19     if (p_tcb == &OSIdleTaskTCB) {                                      (4)
20         CPU_CRITICAL_EXIT();
21         return;
22     }
23
24     /* 时间片自减 */
25     if (p_tcb->TimeQuantaCtr > (OS_TICK)0) {                            (5)
26         p_tcb->TimeQuantaCtr--;
27     }
28
29     /* 时间片没有用完，则退出 */
30     if (p_tcb->TimeQuantaCtr > (OS_TICK)0) {                            (6)
31         CPU_CRITICAL_EXIT();
32         return;
33     }
34
35     /* 如果当前优先级只有一个任务，则退出 */
36     if (p_rdy_list->NbrEntries < (OS_OBJ_QTY)2) {                       (7)
37         CPU_CRITICAL_EXIT();
38         return;
39     }
```

```
40
41      /* 时间片耗完，将任务放到链表的最后一个节点 */
42      OS_RdyListMoveHeadToTail(p_rdy_list);                    (8)
43
44      /* 重新获取任务节点 */
45      p_tcb = p_rdy_list->HeadPtr;                             (9)
46      /* 重载默认的时间片计数值 */
47      p_tcb->TimeQuantaCtr = p_tcb->TimeQuanta;
48
49      /* 退出临界段 */
50      CPU_CRITICAL_EXIT();
51  }
52  #endif  /* OS_CFG_SCHED_ROUND_ROBIN_EN > 0u */
```

代码清单 13-2（1）：时间片是一个可选的功能，是否选择由 OS_CFG_SCHED_ROUND_ROBIN_EN 控制，该宏在 os_cfg.h 定义。

代码清单 13-2（2）：获取链表的第一个节点。

代码清单 13-2（3）：如果节点为空，则退出。

代码清单 13-2（4）：如果节点不为空，看看是否是空闲任务，如果是则退出。

代码清单 13-2（5）：如果不是空闲任务，则时间片计数器 TimeQuantaCtr 减一操作。

代码清单 13-2（6）：时间片计数器 TimeQuantaCtr 递减之后，则判断下时间片是否用完，如果没有用完，则退出。

代码清单 13-2（7）：如果时间片用完，则判断性该优先级下有多少个任务，如果是一个，则退出。

代码清单 13-2（8）：时间片用完，如果优先级下有两个以上任务，则将刚刚耗完时间片的节点移到链表的末尾，此时位于末尾的任务的 TCB 字段中的 TimeQuantaCtr 是等于 0 的，只有等它下一次运行的时候值才会重置为 TimeQuanta。

代码清单 13-2（9）：重新获取链表的第一个节点，重置时间片计数器 TimeQuantaCtr 的值等于 TimeQuanta，任务重新享有时间片。

13.2 修改 OSTimeTick()函数

任务的时间片的单位在每个系统时钟周期到来的时候被更新，时间片调度函数则由时基周期处理函数 OSTimeTick()调用，只需要在更新时基列表之后调用时间片调度函数即可，具体修改见代码清单 13-3 的加粗部分。

代码清单 13-3 OSTimeTick()函数

```
1  void  OSTimeTick (void)
2  {
3      /* 更新时基列表 */
4      OS_TickListUpdate();
5
6  #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
7      /* 时间片调度 */
8      OS_SchedRoundRobin(&OSRdyList[OSPrioCur]);
9  #endif
10
11     /* 任务调度 */
12     OSSched();
13 }
```

13.3 修改 OSTaskCreate()函数

任务的时间片在函数创建的时候被指定，具体修改见代码清单 13-4 中的加粗部分。

代码清单 13-4 OSTaskCreate()函数

```
1  void OSTaskCreate (OS_TCB        *p_tcb,
2                     OS_TASK_PTR   p_task,
3                     void          *p_arg,
4                     OS_PRIO       prio,
5                     CPU_STK       *p_stk_base,
6                     CPU_STK_SIZE  stk_size,
7                     OS_TICK       time_quanta,        (1)
8                     OS_ERR        *p_err)
9  {
10     CPU_STK       *p_sp;
11     CPU_SR_ALLOC();
12
13     /* 初始化 TCB 为默认值 */
14     OS_TaskInitTCB(p_tcb);
15
16     /* 初始化堆栈 */
17     p_sp = OSTaskStkInit( p_task,
18                           p_arg,
19                           p_stk_base,
20                           stk_size );
21
22     p_tcb->Prio = prio;
23
24     p_tcb->StkPtr = p_sp;
25     p_tcb->StkSize = stk_size;
26
27     /* 时间片相关初始化 */
28     p_tcb->TimeQuanta    = time_quanta;             (2)
29 #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
30     p_tcb->TimeQuantaCtr = time_quanta;             (3)
31 #endif
32
33     /* 进入临界段 */
34     OS_CRITICAL_ENTER();
35
36     /* 将任务添加到就绪列表 */
37     OS_PrioInsert(p_tcb->Prio);
38     OS_RdyListInsertTail(p_tcb);
39
40     /* 退出临界段 */
41     OS_CRITICAL_EXIT();
42
43     *p_err = OS_ERR_NONE;
44 }
```

代码清单 13-4（1）：时间片在任务创建的时候由函数形参 time_quanta 指定。

代码清单 13-4（2）：初始化任务 TCB 域的时间片变量 TimeQuanta，该变量表示任务能享有的最大的时间片是多少，该值一旦初始化后就不会变，除非认为修改。

代码清单 13-4（3）：初始化时间片计数器 TimeQuantaCtr 的值等于 TimeQuanta，每经过一个系统时钟周期，该值会递减，如果该值为 0，则表示时间片耗完。

13.4 修改 OS_IdleTaskInit()函数

因为在 OS_IdleTaskInit()函数中创建了空闲任务，所以该函数也需要修改，只需在空闲任务创建函数中，添加一个时间片的形参就可，时间片我们分配为 0，因为在空闲任务优先级下只有空闲任务一个任务，没有其它的任务，具体修改见代码清单 13-5 的加粗部分。

代码清单 13-5 OS_IdleTaskInit()函数

```
1  void  OS_IdleTaskInit(OS_ERR  *p_err)
2  {
3      /* 初始化空闲任务计数器 */
4      OSIdleTaskCtr = (OS_IDLE_CTR)0;
5
6      /* 创建空闲任务 */
7      OSTaskCreate( (OS_TCB     *)&OSIdleTaskTCB,
8                    (OS_TASK_PTR )OS_IdleTask,
9                    (void        *)0,
10                   (OS_PRIO)(OS_CFG_PRIO_MAX - 1u),
11                   (CPU_STK     *)OSCfg_IdleTaskStkBasePtr,
12                   (CPU_STK_SIZE)OSCfg_IdleTaskStkSize,
13                   (OS_TICK        )0,
14                   (OS_ERR      *)p_err );
15 }
```

13.5 main 函数

这里，我们创建任务 1、2 和 3，其中任务的优先级为 1，时间片为 0，任务 2 和任务 3 的优先级相同，均为 2，均分配两个两个时间片，当任务创建完毕后，就绪列表的分布图具体见图 13-2。

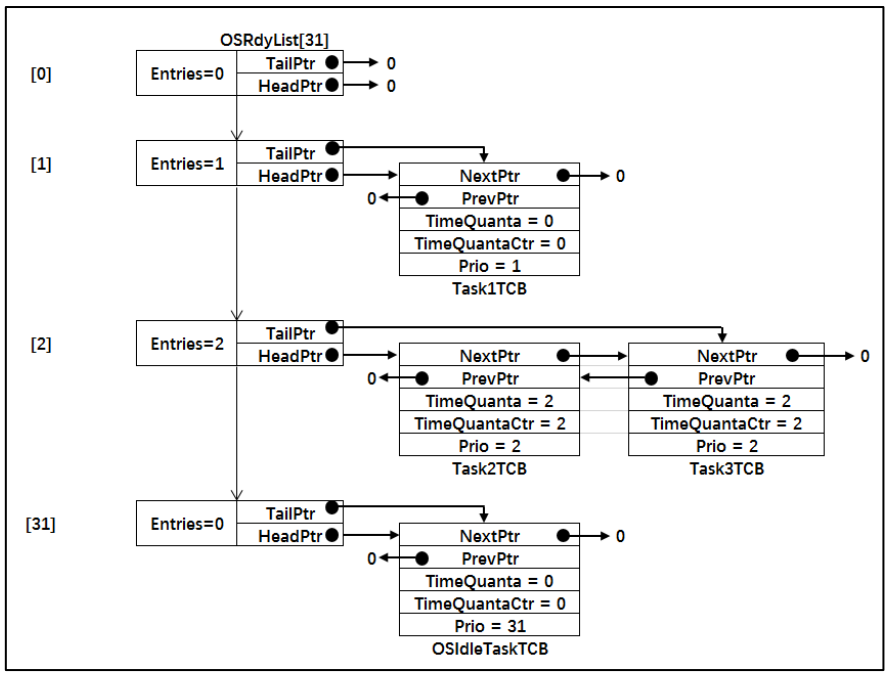


图 13-2 main 函数代码讲解配图

代码清单 13-6 main 函数

```
1  int main(void)
```

```
2  {
3      OS_ERR err;
4
5
6      /* CPU 初始化：1、初始化时间戳 */
7      CPU_Init();
8
9      /* 关闭中断 */
10     CPU_IntDis();
11
12     /* 配置 SysTick 10ms 中断一次 */
13     OS_CPU_SysTickInit (10);
14
15     /* 初始化相关的全局变量 */
16     OSInit(&err);
17
18     /* 创建任务 */
19     OSTaskCreate( (OS_TCB       *)&Task1TCB,
20                   (OS_TASK_PTR   )Task1,
21                   (void         *)0,
22                   (OS_PRIO       )1,                    (1)
23                   (CPU_STK      *)&Task1Stk[0],
24                   (CPU_STK_SIZE  )TASK1_STK_SIZE,
25                   (OS_TICK       )0,                    (1)
26                   (OS_ERR       *)&err );
27
28     OSTaskCreate( (OS_TCB       *)&Task2TCB,
29                   (OS_TASK_PTR   )Task2,
30                   (void         *)0,
31                   (OS_PRIO       )2,                    (2)
32                   (CPU_STK      *)&Task2Stk[0],
33                   (CPU_STK_SIZE  )TASK2_STK_SIZE,
34                   (OS_TICK       )1,                    (2)
35                   (OS_ERR       *)&err );
36
37     OSTaskCreate( (OS_TCB       *)&Task3TCB,
38                   (OS_TASK_PTR   )Task3,
39                   (void         *)0,
40                   (OS_PRIO       )2,                    (2)
41                   (CPU_STK      *)&Task3Stk[0],
42                   (CPU_STK_SIZE  )TASK3_STK_SIZE,
43                   (OS_TICK       )1,                    (2)
44                   (OS_ERR       *)&err );
45
46     /* 启动 OS，将不再返回 */
47     OSStart(&err);
48 }
49
50 void Task1( void *p_arg )
51 {
52     for ( ;; ) {
53         flag1 = 1;
54         OSTimeDly(2);
55         flag1 = 0;
56         OSTimeDly(2);
57     }
58 }
59
60 void Task2( void *p_arg )
61 {
62     for ( ;; ) {
63         flag2 = 1;
64         //OSTimeDly(1);                                  (3)
65         delay(0xff);
66         flag2 = 0;
67         //OSTimeDly(1);
```

```
68          delay(0xff);
69      }
70  }
71
72  void Task3( void *p_arg )
73  {
74      for ( ;; ) {
75          flag3 = 1;
76          //OSTimeDly(1);                                    (3)
77          delay(0xff);
78          flag3 = 0;
79          //OSTimeDly(1);
80          delay(0xff);
81      }
82  }
```

代码清单 13-6（1）：任务 1 的优先级为 1，时间片为 0。当同一个优先级下有多个任务的时候才需要时间片功能。

代码清单 13-6（2）：任务 2 和任务 3 的优先级相同，均为 2，且分配相同的时间片，时间片也可以不同。

代码清单 13-6（3）：因为任务 2 和 3 的优先级相同，分配了相同的时间片，也可以分配不同的时间片，并把阻塞延时换成软件延时，不管是阻塞延时还是软件延时，延时的时间都必须小于时间片。

13.6 实验现象

进入软件调试，点击全速运行按钮就可看到实验波形，具体见图 13-3。在图中我们可以看到，在任务 1 的 flag1 置 1 和置 0 的两个时间片内，任务 2 和 3 都各运行了一次，运行的时间均为 1 个时间片，在这 1 个时间片内任务 2 和 3 的 flag 变量反转了好多次，即任务运行了好多次。

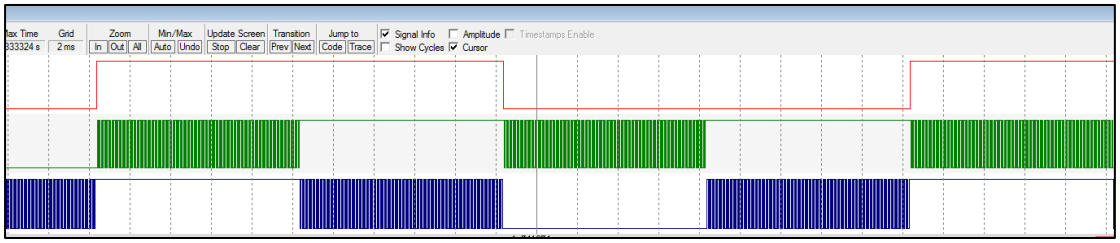


图 13-3 实验现象

第14章　任务的挂起和恢复

本章开始，我们让 OS 的任务支持挂起和恢复的功能，挂起就相当于暂停，暂停后任务从就绪列表中移除，恢复即重新将任务插入到就绪列表。一个任务挂起多少次就要被恢复多少次才能重新运行。

14.1 实现任务的挂起和恢复

14.1.1 定义任务的状态

在任务实现挂起和恢复的时候，要根据任务的状态来操作，任务的状态不同，操作也不同，有关任务状态的宏定义在 os.h 中实现，总共有 9 种状态，具体定义见代码清单 14-1。

代码清单 14-1 定义任务的状态

```
1                                                                  /* ---------- 任务的状态 -------*/
2  #define  OS_TASK_STATE_BIT_DLY              (OS_STATE)(0x01u)  /*   /-------- 挂起位           */
3                                                                  /*   |                        */
4  #define  OS_TASK_STATE_BIT_PEND             (OS_STATE)(0x02u)  /*   | /-----  等待位           */
5                                                                  /*   | |                      */
6  #define  OS_TASK_STATE_BIT_SUSPENDED        (OS_STATE)(0x04u)  /*   | | /---  延时/超时位      */
7                                                                  /*   | | |                    */
8                                                                  /*   V V V                    */
9
10 #define  OS_TASK_STATE_RDY                   (OS_STATE)(  0u)  /*   0 0 0  就绪              */
11 #define  OS_TASK_STATE_DLY                   (OS_STATE)(  1u)  /*   0 0 1  延时或者超时      */
12 #define  OS_TASK_STATE_PEND                  (OS_STATE)(  2u)  /*   0 1 0  等待              */
13 #define  OS_TASK_STATE_PEND_TIMEOUT          (OS_STATE)(  3u)  /*   0 1 1  等待+超时         */
14 #define  OS_TASK_STATE_SUSPENDED             (OS_STATE)(  4u)  /*   1 0 0  挂起              */
15 #define  OS_TASK_STATE_DLY_SUSPENDED         (OS_STATE)(  5u)  /*   1 0 1  挂起 + 延时或者超时 */
16 #define  OS_TASK_STATE_PEND_SUSPENDED        (OS_STATE)(  6u)  /*   1 1 0  挂起 + 等待       */
17 #define  OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED (OS_STATE)(  7u) /*   1 1 1  挂起 + 等待 + 超时 */
18 #define  OS_TASK_STATE_DEL                   (OS_STATE)(255u)
```

14.1.2 修改任务控制块 TCB

为了实现任务的挂起和恢复，需要先在任务控制中 TCB 中添加任务的状态 TaskState 和任务挂起计数器 SusPendCtr 这两个成员，具体见代码清单 14-2 的加粗部分。

代码清单 14-2 任务 TCB

```
1  struct os_tcb {
2      CPU_STK          *StkPtr;
3      CPU_STK_SIZE     StkSize;
4
5      /* 任务延时周期个数 */
6      OS_TICK          TaskDelayTicks;
7
8      /* 任务优先级 */
9      OS_PRIO          Prio;
10
11     /* 就绪列表双向链表的下一个指针 */
12     OS_TCB           *NextPtr;
13     /* 就绪列表双向链表的前一个指针 */
14     OS_TCB           *PrevPtr;
15
16     /*时基列表相关字段*/
17     OS_TCB           *TickNextPtr;
```

```
18      OS_TCB           *TickPrevPtr;
19      OS_TICK_SPOKE    *TickSpokePtr;
20
21      OS_TICK          TickCtrMatch;
22      OS_TICK          TickRemain;
23
24      /* 时间片相关字段 */
25      OS_TICK               TimeQuanta;
26      OS_TICK               TimeQuantaCtr;
27
28      OS_STATE              TaskState;                    (1)
29
30  #if OS_CFG_TASK_SUSPEND_EN > 0u                         (2)
31      /* 任务挂起函数 OSTaskSuspend()计数器 */
32      OS_NESTING_CTR        SuspendCtr;                   (3)
33  #endif
34
35  };
```

代码清单 14-2（1）：TaskState 用来表示任务的状态，在本章之前，任务出现了两种状态，一是任务刚刚创建好的时候，处于就绪态，调用阻塞延时函数的时候处于延时态。本章要实现的是任务的挂起态，再往后的章节中还会有等待态，超时态，删除态等。TaskState 能够取的值具体见代码清单 14-1。

代码清单 14-2（2）：任务挂起功能是可选的，通过宏 OS_CFG_TASK_SUSPEND_EN 来控制，该宏在 os_cfg.h 文件中定义。

代码清单 14-2（3）：任务挂起计数器，任务每被挂起一次，SuspendCtr 递增一次，一个任务挂起多少次就要被恢复多少次才能重新运行。

14.1.3 编写任务挂起和恢复函数

1. OSTaskSuspend()函数

OSTaskSuspend()函数

代码清单 14-3 OSTaskSuspend()函数

```
1  #if OS_CFG_TASK_SUSPEND_EN > 0u
2  void   OSTaskSuspend (OS_TCB  *p_tcb,
3                        OS_ERR  *p_err)
4  {
5      CPU_SR_ALLOC();
6
7
8      #if 0 /* 屏蔽开始 */                                  (1)
9      #ifdef OS_SAFETY_CRITICAL
10     /* 安全检查，OS_SAFETY_CRITICAL_EXCEPTION()函数需要用户自行编写 */
11     if (p_err == (OS_ERR *)0) {
12         OS_SAFETY_CRITICAL_EXCEPTION();
13         return;
14     }
15 #endif
16
17 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
18     /* 不能在 ISR 程序中调用该函数 */
19     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
20         *p_err = OS_ERR_TASK_SUSPEND_ISR;
21         return;
22     }
23 #endif
```

```
24
25      /* 不能挂起空闲任务 */
26      if (p_tcb == &OSIdleTaskTCB) {
27          *p_err = OS_ERR_TASK_SUSPEND_IDLE;
28          return;
29      }
30
31  #if OS_CFG_ISR_POST_DEFERRED_EN > 0u
32      /* 不能挂起中断处理任务 */
33      if (p_tcb == &OSIntQTaskTCB) {
34          *p_err = OS_ERR_TASK_SUSPEND_INT_HANDLER;
35          return;
36      }
37  #endif
38
39  #endif /* 屏蔽结束 */                                              (2)
40
41      CPU_CRITICAL_ENTER();
42
43      /* 是否挂起自己 */                                             (3)
44      if (p_tcb == (OS_TCB *)0) {
45          p_tcb = OSTCBCurPtr;
46      }
47
48      if (p_tcb == OSTCBCurPtr) {
49          /* 如果调度器锁住则不能挂起自己 */
50          if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
51              CPU_CRITICAL_EXIT();
52              *p_err = OS_ERR_SCHED_LOCKED;
53              return;
54          }
55      }
56
57      *p_err = OS_ERR_NONE;
58
59      /* 根据任务的状态来决定挂起的动作 */                           (4)
60      switch (p_tcb->TaskState) {
61      case OS_TASK_STATE_RDY:                                        (5)
62          OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();
63          p_tcb->TaskState  =  OS_TASK_STATE_SUSPENDED;
64          p_tcb->SuspendCtr = (OS_NESTING_CTR)1;
65          OS_RdyListRemove(p_tcb);
66          OS_CRITICAL_EXIT_NO_SCHED();
67          break;
68
69      case OS_TASK_STATE_DLY:                                        (6)
70          p_tcb->TaskState  = OS_TASK_STATE_DLY_SUSPENDED;
71          p_tcb->SuspendCtr = (OS_NESTING_CTR)1;
72          CPU_CRITICAL_EXIT();
73          break;
74
75      case OS_TASK_STATE_PEND:                                       (7)
76          p_tcb->TaskState  = OS_TASK_STATE_PEND_SUSPENDED;
77          p_tcb->SuspendCtr = (OS_NESTING_CTR)1;
78          CPU_CRITICAL_EXIT();
79          break;
80
81      case OS_TASK_STATE_PEND_TIMEOUT:                               (8)
82          p_tcb->TaskState  = OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED;
83          p_tcb->SuspendCtr = (OS_NESTING_CTR)1;
84          CPU_CRITICAL_EXIT();
85          break;
86
87      case OS_TASK_STATE_SUSPENDED:                                  (9)
88      case OS_TASK_STATE_DLY_SUSPENDED:
89      case OS_TASK_STATE_PEND_SUSPENDED:
```

```
90      case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
91          p_tcb->SuspendCtr++;
92          CPU_CRITICAL_EXIT();
93          break;
94
95      default:                                                    (10)
96          CPU_CRITICAL_EXIT();
97          *p_err = OS_ERR_STATE_INVALID;
98          return;
99      }
100
101      /* 任务切换 */
102      OSSched();                                                 (11)
103 }
104 #endif
```

代码清单 14-3（1）和（2）：这部分代码是为了程序的健壮性写的代码，即是加了各种判断，避免用户的误操作。在 uC/OS-III 中，这段代码随处可见，但为了讲解方便，我们把这部分代码注释掉，里面涉及到的一些宏和函数我们均不实现，只需要了解即可，在后面的讲解中，要是出现这段代码，我们直接删除掉，删除掉也不会影响核心功能。

代码清单 14-3（3）：如果任务挂起的是自己，则判断下调度器是否锁住，如果锁住则退出返回错误码，没有锁则继续往下执行。

代码清单 14-3（4）：根据任务的状态来决定挂起操作。

代码清单 14-3（5）：任务在就绪状态，则将任务的状态改为挂起态，挂起计数器置 1，然后从就绪列表删除。

代码清单 14-3（6）：任务在延时状态，则将任务的状态改为延时加挂起态，挂起计数器置 1，不用改变 TCB 的位置，即还是在延时的时基列表。

代码清单 14-3（7）：任务在等待状态，则将任务的状态改为等待加挂起态，挂起计数器置 1，不用改变 TCB 的位置，即还是在等待列表等待。等待列表暂时还没有实现，将会在后面的章节实现。

代码清单 14-3（8）：任务在等待加超时态，则将任务的状态改为等待加超时加挂起态，挂起计数器置 1，不用改变 TCB 的位置，即还在等待和时基这两个列表中。

代码清单 14-3（9）：只要有一个是挂起状态，则将挂起计数器加一操作，不用改变 TCB 的位置。

代码清单 14-3（10）：其它状态则无效，退出返回状态无效错误码。

代码清单 14-3（11）：任务切换。凡是涉及到改变任务状态的地方，都需要进行任务切换。

2. OSTaskResume()函数

OSTaskResume()函数用于恢复被挂起的函数，但是不能恢复自己，挂起倒是可以挂起自己，具体实现见代码清单 14-4。

代码清单 14-4 OSTaskResume()函数

```
1 #if OS_CFG_TASK_SUSPEND_EN > 0u
2 void  OSTaskResume (OS_TCB  *p_tcb,
3                     OS_ERR  *p_err)
4 {
5     CPU_SR_ALLOC();
```

```
 6
 7
 8 #if 0 /* 屏蔽开始 */                                                    (1)
 9 #ifdef OS_SAFETY_CRITICAL
10     /* 安全检查，OS_SAFETY_CRITICAL_EXCEPTION()函数需要用户自行编写 */
11     if (p_err == (OS_ERR *)0) {
12         OS_SAFETY_CRITICAL_EXCEPTION();
13         return;
14     }
15 #endif
16
17 #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
18     /* 不能在 ISR 程序中调用该函数 */
19     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
20         *p_err = OS_ERR_TASK_RESUME_ISR;
21         return;
22     }
23 #endif
24
25
26     CPU_CRITICAL_ENTER();
27 #if OS_CFG_ARG_CHK_EN > 0u
28     /* 不能自己恢复自己 */
29     if ((p_tcb == (OS_TCB *)0) ||
30         (p_tcb == OSTCBCurPtr)) {
31         CPU_CRITICAL_EXIT();
32         *p_err = OS_ERR_TASK_RESUME_SELF;
33         return;
34     }
35 #endif
36
37 #endif /* 屏蔽结束 */                                                   (2)
38
39     *p_err  = OS_ERR_NONE;
40     /* 根据任务的状态来决定挂起的动作 */
41     switch (p_tcb->TaskState) {                                         (3)
42     case OS_TASK_STATE_RDY:                                             (4)
43     case OS_TASK_STATE_DLY:
44     case OS_TASK_STATE_PEND:
45     case OS_TASK_STATE_PEND_TIMEOUT:
46         CPU_CRITICAL_EXIT();
47         *p_err = OS_ERR_TASK_NOT_SUSPENDED;
48         break;
49
50     case OS_TASK_STATE_SUSPENDED:                                       (5)
51         OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();
52         p_tcb->SuspendCtr--;
53         if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
54             p_tcb->TaskState = OS_TASK_STATE_RDY;
55             OS_TaskRdy(p_tcb);
56         }
57         OS_CRITICAL_EXIT_NO_SCHED();
58         break;
59
60     case OS_TASK_STATE_DLY_SUSPENDED:                                   (6)
61         p_tcb->SuspendCtr--;
62         if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
63             p_tcb->TaskState = OS_TASK_STATE_DLY;
64         }
65         CPU_CRITICAL_EXIT();
66         break;
67
68     case OS_TASK_STATE_PEND_SUSPENDED:                                  (7)
69         p_tcb->SuspendCtr--;
70         if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
71             p_tcb->TaskState = OS_TASK_STATE_PEND;
```

```
72          }
73          CPU_CRITICAL_EXIT();
74          break;
75
76      case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:                    (8)
77          p_tcb->SuspendCtr--;
78          if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
79              p_tcb->TaskState = OS_TASK_STATE_PEND_TIMEOUT;
80          }
81          CPU_CRITICAL_EXIT();
82          break;
83
84      default:                                                      (9)
85          CPU_CRITICAL_EXIT();
86          *p_err = OS_ERR_STATE_INVALID;
87          return;
88      }
89
90      /* 任务切换 */
91      OSSched();                                                    (10)
92  }
93  #endif
```

代码清单 14-4（1）和（2）：这部分代码是为了程序的健壮性写的代码，即是加了各种判断，避免用户的误操作。在 uC/OS-III 中，这段代码随处可见，但为了讲解方便，我们把这部分代码注释掉，里面涉及到的一些宏和函数我们均不实现，只需要了解即可，在后面的讲解中，要是出现这段代码，我们直接删除掉，删除掉也不会影响核心功能。

代码清单 14-4（3）：根据任务的状态来决定恢复操作。

代码清单 14-4（4）：只要任务没有被挂起，则退出返回任务没有被挂起的错误码。

代码清单 14-4（5）：任务只在挂起态，则递减挂起计数器 SuspendCtr，如果 SuspendCtr 等于 0，则将任务的状态改为就绪态，并让任务就绪。

代码清单 14-4（6）：任务在延时加挂起态，则递减挂起计数器 SuspendCtr，如果 SuspendCtr 等于 0，则将任务的状态改为延时态。

代码清单 14-4（7）：任务在延时加等待态，则递减挂起计数器 SuspendCtr，如果 SuspendCtr 等于 0，则将任务的状态改为等待态。

代码清单 14-4（8）：任务在等待加超时加挂起态，则递减挂起计数器 SuspendCtr，如果 SuspendCtr 等于 0，则将任务的状态改为等待加超时态

代码清单 14-4（9）：其它状态则无效，退出返回状态无效错误码。

代码清单 14-4（10）：任务切换。凡是涉及到改变任务状态的地方，都需要进行任务切换

14.2 main 函数

这里，我们创建任务 1、2 和 3，其中任务的优先级为 1，任务 2 的优先级为 2，任务 3 的优先级为 3。任务 1 将自身的 flag 每翻转一次后均将自己挂起，任务 2 在经过两个时钟周期后将任务 1 恢复，任务 3 每隔一个时钟周期反转一次。具体代码见代码清单 14-5。

代码清单 14-5 main 函数

```
1  int main(void)
2  {
3      OS_ERR err;
4
```

```
5
6       /* CPU 初始化：1、初始化时间戳 */
7       CPU_Init();
8
9       /* 关闭中断 */
10      CPU_IntDis();
11
12      /* 配置 SysTick 10ms 中断一次 */
13      OS_CPU_SysTickInit (10);
14
15      /* 初始化相关的全局变量 */
16      OSInit(&err);
17
18      /* 创建任务 */
19      OSTaskCreate( (OS_TCB       *)&Task1TCB,
20                    (OS_TASK_PTR   )Task1,
21                    (void         *)0,
22                    (OS_PRIO       )1,
23                    (CPU_STK      *)&Task1Stk[0],
24                    (CPU_STK_SIZE  )TASK1_STK_SIZE,
25                    (OS_TICK       )0,
26                    (OS_ERR       *)&err );
27
28      OSTaskCreate( (OS_TCB       *)&Task2TCB,
29                    (OS_TASK_PTR   )Task2,
30                    (void         *)0,
31                    (OS_PRIO       )2,
32                    (CPU_STK      *)&Task2Stk[0],
33                    (CPU_STK_SIZE  )TASK2_STK_SIZE,
34                    (OS_TICK       )0,
35                    (OS_ERR       *)&err );
36
37      OSTaskCreate( (OS_TCB       *)&Task3TCB,
38                    (OS_TASK_PTR   )Task3,
39                    (void         *)0,
40                    (OS_PRIO       )3,
41                    (CPU_STK      *)&Task3Stk[0],
42                    (CPU_STK_SIZE  )TASK3_STK_SIZE,
43                    (OS_TICK       )0,
44                    (OS_ERR       *)&err );
45
46      /* 启动 OS，将不再返回 */
47      OSStart(&err);
48  }
49
50  void Task1( void *p_arg )
51  {
52      OS_ERR err;
53
54      for ( ;; ) {
55          flag1 = 1;
56          OSTaskSuspend(&Task1TCB,&err);
57          flag1 = 0;
58          OSTaskSuspend(&Task1TCB,&err);
59      }
60  }
61
62  void Task2( void *p_arg )
63  {
64      OS_ERR err;
65
66      for ( ;; ) {
67          flag2 = 1;
68          OSTimeDly(1);
69          //OSTaskResume(&Task1TCB,&err);
70          flag2 = 0;
```

```
71          OSTimeDly(1);;
72          OSTaskResume(&Task1TCB,&err);
73      }
74  }
75
76  void Task3( void *p_arg )
77  {
78      for ( ;; ) {
79          flag3 = 1;
80          OSTimeDly(1);
81          flag3 = 0;
82          OSTimeDly(1);
83      }
84  }
```

14.3 实验现象

进入软件调试，点击全速运行按钮就可看到实验波形，具体见图 14-1。在图 14-1 中，可以看到任务 2 和任务 3 的波形图是一样的，任务 1 的波形周期是任务 2 的两倍，与代码实现相符。如果想实现其它效果可自行修改代码实现。

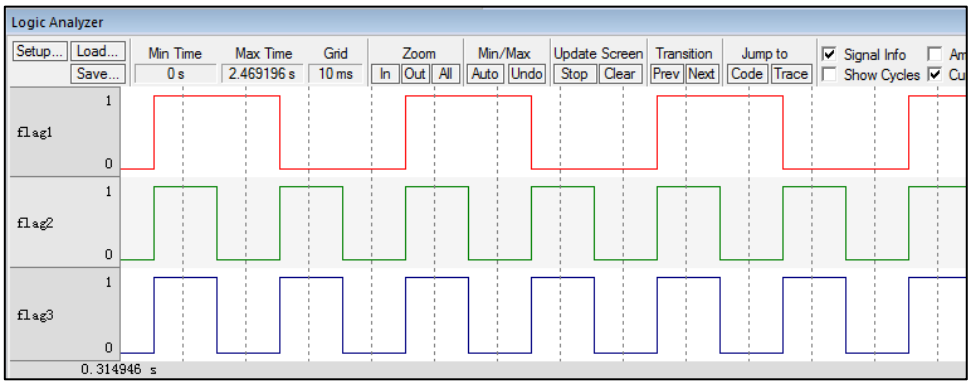


图 14-1 实验现象

第15章　任务的删除

本章开始，我们让 OS 的任务支持删除操作，一个任务被删除后就进入休眠态，要想继续运行必须创新创建。

15.1 实现任务删除

15.1.1 编写任务删除函数

1. OSTaskDel()函数

任务删除函数 OSTaskDel()用于删除一个指定的任务，也可以删除自身，在 os_task.c 中定义，具体实现见代码清单 15-1。

代码清单 15-1 OSTaskDel()函数

```
1  #if OS_CFG_TASK_DEL_EN > 0u                                          (1)
2  void  OSTaskDel (OS_TCB  *p_tcb,
3                   OS_ERR  *p_err)
4  {
5      CPU_SR_ALLOC();
6
7      /* 不允许删除空闲任务 */                                          (2)
8      if (p_tcb == &OSIdleTaskTCB) {
9          *p_err = OS_ERR_TASK_DEL_IDLE;
10         return;
11     }
12
13     /* 删除自己 */
14     if (p_tcb == (OS_TCB *)0) {                                      (3)
15         CPU_CRITICAL_ENTER();
16         p_tcb  = OSTCBCurPtr;
17         CPU_CRITICAL_EXIT();
18     }
19
20     OS_CRITICAL_ENTER();
21
22     /* 根据任务的状态来决定删除的动作 */
23     switch (p_tcb->TaskState) {
24     case OS_TASK_STATE_RDY:                                          (4)
25         OS_RdyListRemove(p_tcb);
26         break;
27
28     case OS_TASK_STATE_SUSPENDED:                                    (5)
29         break;
30
31     /* 任务只是在延时，并没有在任何等待列表*/
32     case OS_TASK_STATE_DLY:                                          (6)
33     case OS_TASK_STATE_DLY_SUSPENDED:
34         OS_TickListRemove(p_tcb);
35         break;
36
37     case OS_TASK_STATE_PEND:                                         (7)
38     case OS_TASK_STATE_PEND_SUSPENDED:
39     case OS_TASK_STATE_PEND_TIMEOUT:
40     case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
41         OS_TickListRemove(p_tcb);
```

```
42
43 #if 0 /* 目前我们还没有实现等待列表，暂时先把这部分代码注释 */
44         /* 看看在等待什么 */
45         switch (p_tcb->PendOn) {
46         case OS_TASK_PEND_ON_NOTHING:
47         /* 任务信号量和队列没有等待队列，直接退出 */
48         case OS_TASK_PEND_ON_TASK_Q:
49         case OS_TASK_PEND_ON_TASK_SEM:
50             break;
51
52         /* 从等待列表移除 */
53         case OS_TASK_PEND_ON_FLAG:
54         case OS_TASK_PEND_ON_MULTI:
55         case OS_TASK_PEND_ON_MUTEX:
56         case OS_TASK_PEND_ON_Q:
57         case OS_TASK_PEND_ON_SEM:
58             OS_PendListRemove(p_tcb);
59             break;
60
61         default:
62             break;
63         }
64         break;
65 #endif
66     default:
67         OS_CRITICAL_EXIT();
68         *p_err = OS_ERR_STATE_INVALID;
69         return;
70     }
71
72     /* 初始化 TCB 为默认值 */
73     OS_TaskInitTCB(p_tcb);                                          (8)
74     /* 修改任务的状态为删除态，即处于休眠 */
75     p_tcb->TaskState = (OS_STATE)OS_TASK_STATE_DEL;                  (9)
76
77     OS_CRITICAL_EXIT_NO_SCHED();
78     /* 任务切换，寻找最高优先级的任务 */
79     OSSched();                                                      (10)
80
81     *p_err = OS_ERR_NONE;
82 }
83 #endif /* OS_CFG_TASK_DEL_EN > 0u */
```

代码清单 15-1（1）：任务删除是一个可选功能，由 OS_CFG_TASK_DEL_EN 控制，该宏在 os_cfg.h 中定义。

代码清单 15-1（2）：空闲任务不能被删除。系统必须至少有一个任务在运行，当没有其它用户任务运行的时候，系统就会运行空闲任务。

代码清单 15-1（3）：删除自己。

代码清单 15-1（4）：任务只在就绪态，则从就绪列表移除。

代码清单 15-1（5）：任务只是被挂起，则退出返回，不用做什么。

代码清单 15-1（6）：任务在延时或者是延时加挂起，则从时基列表移除。

代码清单 15-1（7）：任务在多种状态，但只要有一种是等待状态，就需要从等待列表移除。如果任务等待是任务自身的信号量和消息，则直接退出返回，因为任务信号量和消息是没有等待列表的。等待列表我们暂时还没实现，所以暂时将等待部分相关的代码用条件编译屏蔽掉。

代码清单 15-1（8）：初始化 TCB 为默认值。

代码清单 15-1（9）：修改任务的状态为删除态，即处于休眠。

代码清单 15-1（10）：任务调度，寻找优先级最高的任务来运行。

15.2 main 函数

本章 main 函数没有添加新的测试代码，只需理解章节内容即可。

15.3 实验现象

本章没有实验，只需理解章节内容即可。

第16章　IPC 通信

本章之前，uC/OS-III 的核心部分已经完成，剩下的就是 IPC 部分，如信号量、互斥量、队列、事件标志组，任务信号量，任务消息等。这部分的程序已经写好，但是文档部分今年是写不完了，需要明年了，暂时先放出第一部分给大家阅读。

附录 uC/OS 的编程风格

文件头

文件包含

标识符命名法

缩写及助记符

注释

定义语句

数据类型

局部变量

函数原型

函数声明

缩进格式

语句和表达式

语句和表达式应该在一行中写完，每行不多于一个赋值语句，不要用下面的写法。

```
1 x=y=z=1;
```

虽然在 C 语言中，这种写法是被允许的。但是，当变量名变得更加复杂时，程序的意图会变得不明显。

以下运算符的前后都不要加空格。

“->”，结构体指针运算符，p->m

“.”，结构体成员运算符，s.m

“[]”，数组下标，a.[i]

函数和函数后面的括号之间不要加空格；逗号后面要加一个空格，用于将函数参数列表中的参数分开；括号中的表达式前后不要加空格；前括号后面与后括号前面也不要加空格；逗号和分号的后面应该加一个空格。

```
Strncat(t, s, n);
for (i=0; i<n; i++)
```

一元运算符与其操作数之间不要加空格。

```
!p, -b, ++i, --j, (loog)m, *p, &x, sizeof(k)
```

二/三元运算符的前面和后面可以加一个或者几个空格。

```
c1 = c2, x + y, i+=2, n>0 ? n : -n
```

关键字 if，while，for，switch，以及 return 后面要加一个空格。

对于赋值语句，数值要按列对齐，符号也要上下对齐。

```
x           = 100.567;
temp        = 12.400;
var5        = 0.768;
variable    = 12;
storage     = &array[0];
```

结构体和联合体