提供源码，讲述原理

从无到有，和你一起一步步编写实时嵌入式操作系统内核

操作系统内核也许并没有像你想象的那么神秘

底层工作者手册

之嵌入式操作系统内核

Wanlix 操作系统内核
Mindows 操作系统内核

我在写本手册前没有任何有关操作系统内核的知识，有的仅仅是简单的使用过 2 个操作系统的经验，也仅限于对操作系统应用层一些功能的简单了解。我在写操作系统内核时也只能从这些应用经验出发，参考一些资料，更多的是自己想办法用最顺其自然的代码实现操作系统的这些内核功能。因此，你要相信，既然我能在此基础上写出这个操作系统内核那么你一定也能看明白。

本手册不仅仅是从应用的角度介绍操作系统如何使用，更重要的是从原理的角度对操作系统的功能做了分析、设计，从无到有循序渐进一点点的增加操作系统的功能，并且每增加一个功能便配以一个例子加以演示，让读者能立刻看到代码运行的结果。

本手册记录了我从对操作系统内核不了解到写出操作系统内核的过程，这样的一个过程对你来说应该也是一个最好的学习过程。

如果你有一定的 C 语言基础，并且对硬件也有稍微的了解，那么我相信你一定会看明白本手册！也一定可以随心所欲的修改、扩展你需要的操作系统功能！

书并不只是简单的翻译文档
书可以写的让人看得更明白

前言

目前我所见的绝大部分介绍操作系统的书籍只是从应用的角度告诉读者应该如何使用操作系统，而且相当一部分书籍只是把原有的用户手册整理了一下便出书了，这样的书籍只能当做一本使用手册去查，从学习的角度来说意义不大，一不介绍实现背景、原理，二不介绍应用例子，无法让读者深刻体会操作系统的用法。本手册最大特点是从操作系统的结构设计、编码的角度讲述操作系统内核原理。本手册不是在操作系统写完后才写的，而是一边设计一边编码一边编写，记录了操作系统从无到有的过程，讲解了操作系统实现的原理，只要读者了解 C 语言，再对汇编语言和硬件稍微有所了解便能看懂本手册。

05 年 4 月，经历了漫长的学生时代我终于参加工作了！

在学校里接触了少的可怜的硬件开发，由于无人指导再加上本人做和尚撞钟，因此所调试的单板问题百出。进入公司后，当我可在硬件与底层软件之间选择时我毫不犹豫的选择了软件，直至走到今天。最开始被分配到做微码，后来又阴错阳差的搞起了 C 语言底层软件开发。我刚入公司时可谓软件基础太差，学校里学的知识也使我仅知道一点 C 语言的概念，从来没有实战过。好在当时所作的项目编码阶段已经结束，我的工作就是学习别人的代码并帮助测试、修改问题，当然，做的也并不好。现在回想起来，在这平淡的工作过程中有三点对我至关重要，一、正是在这段时间培养起我比较扎实的 C 语言基础，不能说学到了很多，但绝对是让我明白了很多最基本的概念，让我知道了学习的方法。二、正是在这段时间我接触了项目的开发，让我参与到历时几年几百人相互协助的项目开发中，看到大项目的开发过程，接触到了很多在学校里永远不会接触到的事物，这些经验对我今后至关重要，虽然只是冰山一角。三、正是在这段时间让我有机会第一次接触了嵌入式操作系统——vxworks，虽然仅仅是嵌入式操作系统的一些应用层概念。

由于我基础较差再加上我是慢热型，当时工作的并不好，一年半后几经周折我换到了一个小部门。以前几百人的开发团队不见了，众多的技术专家、牛人不见了，一二十层、几个、几十个 CPU 的板子不见了，取而代之的是巴掌大的单板，所谓专家就是我，我一个人就可以是整个项目的全部软件开发人员，设计软件结构、编写从驱动层到业务层的所有代码。以前所做的工作是冰山一角，只知功能不识业务，现如今则需要我承担与软件相关的所有工作。正是在这种环境中我可以借鉴以前的一些经验并按照自己现有的想法设计软件，在实现系统功能的同时也证明了我在硬件、底层软件所掌握知识的正确性。从做大系统的冰山一角，到做麻雀虽小五脏俱全的小系统，各有各的难处，但也各有各的优点，这也为我编写这本手册提供了必要条件。

在做这些小系统时有一个问题一直困扰着我，我所作的设备需要与主设备对接，主设备会实时下发命令给我们执行，并且需要实时回应消息，这样看来如果有一个嵌入式操作系统就会比较好实现。但我们的小系统硬件资源受限制，主频低、存储空间少，使得我很难找到一个合适的操作系统。现有的一些能用的操作系统需要收费，有些不提供源码，但让我最不能接受的是资料不全，真看不明白，使用这些操作系统如果在项目开发过程中出了问题又没有很好的技术支持将是很大的风险，因此在做这些小系统时我一直是裸奔。裸奔是可以搞定一切，但对于系统设计、维护来说确实是比较费劲。

在一个项目中我抛弃了原有的 51 单片机，使用了 ARM7TDMI 处理器。随着反复查看 ARM 芯片手册并在项目调试过程中对 ARM7 芯片的逐步了解，我逐渐意识到实现一个简单的操作系统内核调度功能似乎并没有想象中的那么困难，原以为实现操作系统调度功能需要

深入了解编译器的知识，现在发现只要使用标准的 C 语言、一些汇编语言和芯片硬件知识就可以实现。

整理一下我目前所处的情况：

1. 迫切需要一个适合小系统的嵌入式操作系统，但又没有合适的。
2. 了解了嵌入式操作系统的一些概念。
3. 掌握了 ARM7 芯片的硬件结构、C 语言和汇编语言知识。
4. 找不到一本可以较好的介绍操作系统的书籍，希望能让更多的人了解嵌入式操作系统内核调度的基本原理，并以一种简单易懂的方式让更多的人接受。

事已如此，万事具备！现在，我们就开始一起编写两个嵌入式操作系统内核——Wanlix 和 Mindows！

Wanlix 是一个内核非常小的嵌入式操作系统，只有几百个字节（大小与编译器、编译选项也有关），但功能也非常少，只提供任务切换功能，而且需要主动调用函数切换任务。但，它确实可以实现任务调度功能，最难能可贵的是它的小巧，非常适合资源特别少但又需要任务切换的小项目。在这个源码开放的时代，Linux、Unix 遍地生根，它就跟我姓了，因此叫 Wanlix。

地球人都知道 Windows，它是一种大型 PC 机操作系统，它是分时操作系统，它是 PC 机通用操作系统。而我们将要编写的 Mindows 则是一种小型操作系统，是实时的，是用在嵌入式设备上的嵌入式实时操作系统，一切都是与 Windows 相反的！因此这个操作系统就叫 Mindows！

本手册只讲解 Wanlix、Mindows 操作系统的内核，至于其它的例如 BSP、文件系统、协议栈等内容过于庞大，本人没有精力也没有能力实现。这两个操作系统已经提供了源码，有兴趣的朋友可以在此基础上自己试着实现其它功能，与他人互相讨论、交流，共同提高。在此我为大家提供了一个网站：

http://blog.sina.com.cn/ifreecoding

大家可以登录此网站下载相关资料，并可进入其中的论坛交流经验。

本手册假定读者具有一定的软硬件基础，对于其中软件编码方面的基础问题不再赘述。

另外需要特殊说明的是，我使用 vxworks 嵌入式操作系统时间只有一年左右，而且只是使用过极其简单的几个最基本的功能，在后来的一个项目中还简单使用过 TI DSP 的 BIOS 操作系统，因此本人对嵌入式操作系统的了解仅限皮毛，本手册也仅是根据本人在使用上述两种操作系统中所建立的感官印象并按照我自己的想法来实现的，错误、疏漏之处在所难免，还请各位多多包涵，如有问题，可以反馈到论坛。

本人免费提供 Wanlix 和 Mindows 的源码，但不承担您使用本操作系统为您带来的损失。

另外，本人语文水平实在有限，当我还年轻的时候就因为高中还需要写作文，就没有报考高中，后来是班主任硬逼着改报的高中，在此向当年的班主任孙老师表示感谢！因此，本手册无法顾及语言优美逻辑顺通，只要大家能看明白就行了，有问题我们可以再交流。

最后，向那些无偿付出自己知识的兄弟姐妹们表示敬意！在编写操作系统过程中，确实遇到了一些问题，正是在网上查到你们贡献出的宝贵经验才能让我得以完成此操作系统的编写，因此，我也将这本手册无偿提供给大家，供大家参考，希望本手册能给你能带来一些帮助！

2011.09.23 深圳坂田

目录

图

表

第 1 章 操作系统基础知识

有很多嵌入式系统设备的资源非常少，几十 K 的 ROM，几 K 的 RAM，这种小系统设备上的软件功能也非常简单，软件只要按照设定好的功能周而复始的运行就可以了。这种小系统设备不需要操作系统，也几乎没有合适的操作系统能运行在资源如此少的设备上。

当芯片资源越来越丰富，要实现的功能越来越多的时候，你就会发现软件所做的工作不再是简单的重复一件事情了，它需要及时的响应外部的输入信号，需要及时协调自己内部的运行状态，而且多个功能的软件可能会同时运行在一套硬件资源上，这样，软件不能只是简单的按照自己的计划完成自己的事情就可以了，它还需要不断的与外界交互，及时满足其它要求，并根据其它的要求及时调整自己的状态。

本章将从几个例子开始，说明在没有操作系统的情况下软件编程的不便之处，以帮助读者理解使用操作系统的任务管理功能，并通过介绍操作系统的相关概念使读者对操作系统有一个基本了解，在后面的章节将依靠这些知识，先实现一个非常简单小巧的非抢占操作系统内核——Wanlix，然后再实现一个实时抢占操作系统内核——Mindows。

第 1 节 为什么要使用操作系统

在没有操作系统的情况下，C 语言是以函数为单位实现功能的，一个函数一个函数串行的执行，一个完整的功能会由多个函数共同完成。然而当软件系统的功能变得多而庞大的时候，这种方法几乎无法使用，因为此时各个功能之间必然会有千丝万缕的联系，不可能依次串行的完成每个功能，各个功能必然需要交替执行。以函数为功能单元的程序很难在执行一个函数的时候转而去执行另外不相关的函数，即使是使用一些技巧实现了，也会使整个软件结构变的混乱不堪，不利于软件的维护和扩展。函数的工作方式就决定了并不适合以它为功能单元运行复杂的程序，在这种情况下就需要使用操作系统了。操作系统是对函数运行管理的系统，它可以在一个函数还没有运行完就转而去执行另外一个函数，并且还可以恢复到原来的函数继续执行，这样就可以根据需要及时调整到需要运行的函数来满足各种要求。

以大家熟悉的 Windows 为例，Windows 上运行了很多软件，有办公的、看电影的，玩游戏的，等等等等，太多了。你想过没有，它们是怎么运行的？它们是由不同的厂商开发的，它们之间如何协调？谁先运行谁后运行？这些就是操作系统要做的事。这些应用程序从宏观上看是在一台电脑上同时运行，但从微观上看它们是串行运行的。电脑的 CPU 每一时刻只能运行一个应用程序，运行很短的时间之后，CPU 又去运行下一个应用程序，周而复始的这么运行。由于 CPU 的速度特别快，因此每个应用程序在很短的时间都可以运行很多次，以人的感觉来说，根本就感觉不到 CPU 在各个应用程序之间切换运行，因此我们就觉得电脑上的每个应用程序都是在同时运行。就像看电影一样，由于影片的刷新频率快过了人眼睛的可分辨频率，因此我们就觉得电影是在连续播放。这就是操作系统的一个重要功能——任务调度功能。

除此之外，操作系统还有很多功能，比如说文件系统。我们存储的游戏、电影文件是如何放在硬盘上的？为什么我们将几 G 的文件剪切到同一个硬盘分区上时间很短，而剪切到另外一个硬盘分区上则时间很长？为什么在 Dos6 下看不到 NTFS 分区的文件？这些都是操作系统的一个功能——文件管理功能。

另外，操作系统还具有设备管理功能。现在我们在 Windows 环境下，可以把一块显卡、声卡直接插到主板上，然后启动电脑，安装驱动程序，甚至不需要安装驱动程序就可以使用了。你可能认为电脑就应该是这样的，但实际上，这简单的背后是操作系统为我们做了很多工作，在过去操作系统并不完善的日子，我们需要手动为硬件分配物理地址、中断等资源，极其麻烦。

一个完整的操作系统应该是一个非常复杂非常庞大的系统，还需要包含很多其它的功能，但由于本人能力及精力有限，这些不在本手册的讲述之中，本手册将重点介绍实现嵌入式操作系统的内核调度功能，只侧重任务调度部分，编写一个操作系统内核，读者如有兴趣可自行在此基础上实现操作系统更多的功能。

对于功能简单的小系统设备来说，我们只需要设计一个 while 死循环就可以完成所有的软件功能，这种小系统一般没有复杂的外部输入，例如电子表，外部输入只有调节时间的按钮，软件的主要功能也只是读取定时器的数值并显示出来。我们以伪码的形式描述一个这样的软件结构：

```
int main(void)
{
    while(1)
    {
        1.判断按键输入并执行相关操作。
        2.读取定时器数值。
        3.刷新液晶屏显示时间。
    }
}
```

这个小系统的运行几乎不依赖于外界的输入，只要按照软件设定好的顺序周而复始的执行就可以实现所有功能。

但如果系统功能复杂一些统，使用上述的软件结构就显得有些不适合了。例如我们常用的手机，一般手机处于不通话的状态下，屏幕是黑的，但这并不代表软件没有工作，此时软件需要检测按键是否被按下，闹表定时是否到了，是否有电话来了等等，假设用户在使用手机上网，同时又在听音乐，而电话又来了，你想想软件这个 while 循环应该如何去写？手机中软件遇到的情况可要比我列出的上述情况复杂的多，仅仅使用这个 while 循环是无法完成的。

从手机的例子我们可以看到一个软件系统是由多个功能组成的，有些功能之间相对比较独立，例如听音乐与上网是没什么联系的，发短信与闹表是没什么联系的。因此，我们很容易想到，如果软件是以功能为单位去运行的，而各个功能又可以同时运行，那么每个功能只需要专注完成自己的功能就可以了，上述这么复杂的问题也就可以迎刃而解了。

但传统的函数调用方式无法同时运行多个功能函数，因此，我们就无法使用传统的函数方式同时执行多个功能。

前面我们说过，操作系统从宏观来看是可以实现多个功能同时运行的，这种宏观的同时运行是建立在微观的从一个函数的运行过程切换到另一个函数的运行过程实现的，并且还可以再切换回原来的函数继续运行。这种在函数间跳来跳去的运行方式就是操作系统赖以生存的最核心功能——系统调度功能。从原理上来说，这个实现过程并不复杂，并且只需要使用 C 语言和一点点汇编语言外加一点点技巧就可以实现，所用的软件与我们平时编程时用的软件没什么区别。

操作系统是以任务为执行单元的，每个任务就是一个相对独立的功能单元，各个任务之间可以并行运行，因此操作系统也就实现了多个功能的并行运行功能。每个任务是使用一个函数创建的，没有操作系统的函数和操作系统中创建任务的函数是没有什么区别的，主要区

别在于操作系统可以使用一些技巧，让以任务形式存在的函数可以在运行时互相切换。当然，为了实现这个功能，还需要为创建操作系统的函数增加一些额外的属性，将函数变成任务，这个我们将会在后面章节讲述。

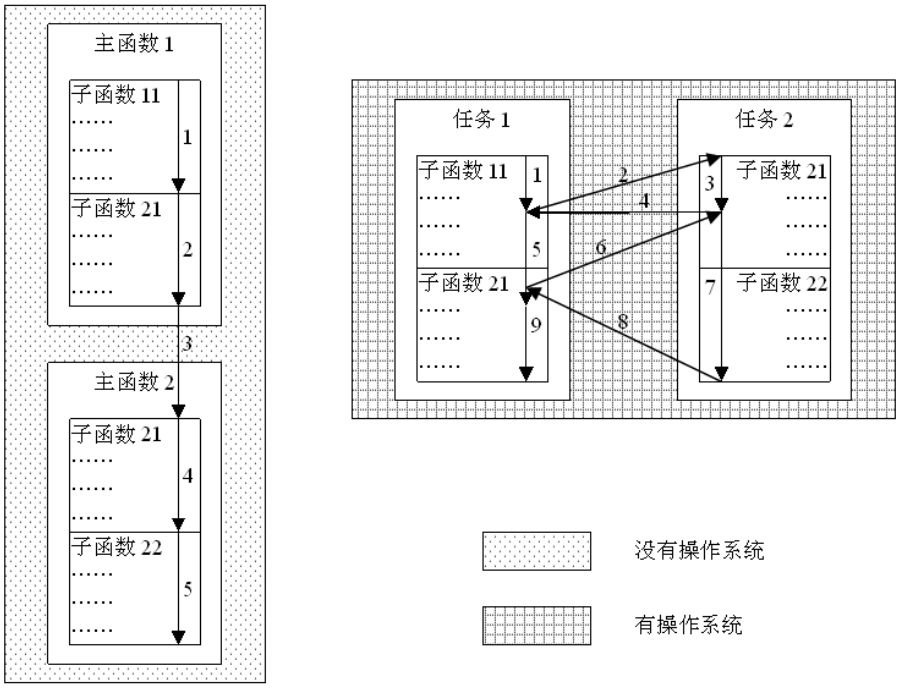


图 1　没有操作系统和有操作系统的函数执行过程

正是任务具有同时执行的特点，我们可以将几个不相关的功能分别用几个任务来实现，例如手机的听音乐、上网、发短信和闹表等功能，我们使用操作系统为每个功能建立一个任务，每个任务的代码只重点关心自己的功能，至于任务间的间插执行就交给操作系统了，这样就使得整个软件结构变的清晰简单。

第 2 节 操作系统的分类

操作系统是管理整个软硬件系统的软件系统，从不同的角度操作系统可以有很多种划分，比如按与用户对话的界面分类可分为命令行界面操作系统和图形用户界面操作系统，按支持用户数的多少可以分为单用户和多用户操作系统，按功能可以分为嵌入式操作系统和 PC 机通用操作系统，按调度的方式可分为分时系统和实时系统等。操作系统种类繁多，很难用单一标准统一分类，由于本人知识有限无法详细的介绍各种类型操作系统，也无法为操作系统准确分类。对比 PC 机使用的操作系统，本手册将讲述的是嵌入式实时操作系统，因此将介绍一下“嵌入式”和“实时”等概念。

◆ 嵌入式操作系统（Embedded Operating System，EOS）

根据 IEEE（The Institute of Electrical and Electronics Engineers，电气与电子工程师学会）的定义，嵌入式系统是控制、监视或者辅助装置、机器和设备运行的装置（devices used to control, monitor, or assist the operation of equipment, machinery or plants）。从中可以看出嵌入式系统是软件和硬件的结合体，按我个人的理解，嵌入式软件就是“嵌入”到硬件中的软件，

而嵌入到硬件中的操作系统就是嵌入式操作系统。这个“嵌入”是相对 PC 机而言的，PC 机是一个通用的系统，有着标准的外设定义，键盘、鼠标、显示器、显卡、声卡、各种标准的插槽，x86 的 CPU，买台电脑功能都差不多，差的只是性能。而嵌入式设备则五花八门，PSP、MP4、手机、电子称、遥控器等等，什么都有，它们的硬件系统是针对专一功能开发的，它们的软件和操作系统也具有专一性，因此体积小成本低。

我们对比一下使用嵌入式系统和 PC 机通用系统开发产品，举个例子，如果要做一个计算器，我这里有两个方案，一、用电脑做，买来电脑，装完 Windows，在运行窗口敲入“calc”，可以直接调出计数器软件，功能实现了。优点是开发周期短，而且 PC 机上也有众多的软件可以使用，扩展性强。但缺点也是致命的，成本太高体积太大，不能指望着小商小贩们背着电脑去卖货，这样的产品一定卖不出去。二、使用单片机、LED 显示屏等器件自己设计方案开发产品，虽然开发周期相对要长一些，但成本绝对低。再举个例子，如果要开发一种功能丰富的办公系统产品，则最好是基于 PC 机系统开发的。键盘、鼠标、显示器、打印机、扫描仪、传真机、摄像头，这些办公常用的输入输出设备与 PC 机都有标准的接口，可以直接使用，而且 PC 机上丰富的软件可以使开发过程容易很多，如果自己另做一套软硬件，这个工作量太大了，几乎无法完成，而且这么大的工作量也会使成本居高不下。

本手册所实现的两个操作系统——Wanlix 和 Mindows 都属于嵌入式操作系统，这两个操作系统在设计时都定位为小系统的操作系统，因此具有内核小的特点。Wanlix 的内核非常小，定位于非常低端的软硬件系统，Mindows 可提供多种操作系统功能，用户也可根据自身需求选取需要的部分，也可在此基础上编写代码增加自己需要的功能，具有可裁剪性。

◆ 实时操作系统（Real-time Operating System，RTOS）

实时是指及时性，实时操作系统具有实时性，能保证及时做出响应。某些领域对数据采集、处理的实时性要求比较严格，时间上的错误可能会造成灾难性的后果，因此需要软件具有很高的实时处理能力。操作系统是控制软件运行的系统，为实现软件的实时性就需要操作系统具有实时性，实时操作系统可以快速响应外界及内部状态的变化，在严格规定的时间内完成相关工作的调度，具有高可靠性。与之相对的分时操作系统则按时间片依次逐个调度任务，实时性不高。实时操作系统是一种抢占式操作系统（Preemptive operating system），所谓抢占式是指高优先级的任务可以中断正在运行的低优先级任务，处理器转而去执行高优先级的任务，由于这种“抢占”可在高优先级任务就绪后立刻发生，因此才保证了操作系统的实时性。

Wanlix 是非抢占式操作系统，需要由当前运行的任务主动发起任务切换调度，其它任务不可中断其运行，因此实时性不高。Mindows 是实时抢占式操作系统，任务支持多种优先级抢占调度，将实时性高的任务设置为高优先级就可以保证软件系统的实时性。

第 2 章 写操作系统前的预备知识

通过前面章节的介绍我们对操作系统有了初步的了解，但这也只是停留在概念阶段，这些知识对于写一个操作系统来说是远远不够的。从现在的章节开始，我们将从无到有，一步一步一个功能一个功能的写出操作系统。

本章我们就先了解一下写操作系统所需要的知识，会涉及到一些汇编语言及芯片的内部结构，如果你没有这方面相关的基础的话，看起来可能会枯涩难懂一些，如果是这种情况的话，建议粗略看一下就可以了，不要过分追求细节。

我们首先将在 ARM7 芯片上编写操作系统，因此本章将对 ARM7 芯片的内部结构做一些介绍，并会介绍一下相关的汇编语言，以及 C 语言与汇编语言之间的关系，最后再介绍一下 Wanlix 操作系统的文件组织结构及开发环境。

第 1 节 ARM7 芯片基本结构

ARM7 芯片构架比较简单，32bits 线性地址空间统一排列，任何地址都是唯一的，不同的片上资源及外设被分配到不同的地址空间，不同数据结构的指针固定为 4 字节长度，这相对 51 芯片来说方便很多也清晰很多，从用户编程的角度来看入手比较简单，因此本手册首先选用 ARM7 芯片来作为开发操作系统的硬件平台。选用的 ARM7 芯片，是 ADI 公司的一款芯片——Aduc7024。

Aduc7024 具有片上 AD、DA、GPIO、UART、I2C、SPI、TIMER、WDT、PWM 等外设，具有 62KBytes 的内部 FLASH 程序空间和 8KBytes 的内部 RAM 空间，无需外挂 ROM 和 RAM，芯片的具体细节可以查阅附录中的参考文档 1。在我们的开发过程中，我们主要使用了芯片的 UART，也就是串口，作为打印数据的端口，输出到 PC 上来观察操作系统的运行。

当我们完成了操作系统的一些基本功能后，我们会将 Wanlix 和 Mindows 移植到另外一款 ARM 芯片上——Cortex 内核的 ARM 芯片，并在此芯片上继续完善 Mindows 的功能。这么做，第一是让读者了解 wanlix 和 Mindows 的移植过程，第二是让读者在移植过程中体会操作系统与用户代码无关的重要性，第三，Cortex 内核芯片功能更强、资源更丰富，可以在此芯片上实现更多的功能。Cortex 内核是 ARM 公司新推出的一种内核，其功能强大，性价比高，后面章节我们再详细讨论，我们首先来了解 ARM7 芯片。

ARM7 支持 7 种处理器模式，分别是 USR 模式、SYS 模式、SVC 模式、ABT 模式、UND 模式、IRQ 模式和 FIQ 模式，虽然有这么多种模式但可以归纳为 2 大类：正常工作模式和中断工作模式。

正常工作模式包括 USR 模式和 SYS 模式，USR 模式没有任何特性，是芯片最常用的模式，芯片一般都是在 USR 模式下运行的。SYS 模式与 USR 模式也没什么区别，仅比 USR 模式权限大些，能访问到芯片的特殊寄存器，适合操作系统使用（但我不知道怎么使用，本手册没有使用该模式）。

而中断工作模式又分为异常中断模式和正常中断工作模式。芯片无法正常运行时就会进入异常中断模式，芯片进入异常中断模式后软件就无法再继续提供原有功能了，异常中断模

式仅是为定位问题而提供的。异常中断模式包括 ABT 模式和 UND 模式，指令或数据出错时就会进入 ABT 模式，例如，ARM 模式下的一条指令是需要访问 4 字节对齐的地址，如果实际访问的地址不是 4 字节对齐的话，这时芯片就会进入 ABT 模式，程序 PC 指针就会跳转到 ABT 的异常中断向量，执行 ABT 的中断服务程序。如果用户为 ABT 模式编写了定位信息程序并挂到 ABT 异常中断上，那么就可以利用这些代码输出产生 ABT 异常的原因了。UND 模式与 ABT 模式的工作原理是一样的，只不过 UND 模式是在遇到没有定义的指令时才会进入。

正常中断模式是为用户实现系统功能而设计的中断模式，它是预先设计好的，是系统运行所必须的。正常中断模式又分为 IRQ 模式、FIQ 模式和 SVC 模式。IRQ 模式就是普通的中断模式，当芯片产生中断时就会进入 IRQ 模式，等同于其它芯片的中断。FIQ 是快速中断模式，它比 IRQ 中断优先级高，备份、还原中断现场时间更少些，也就是说能更优先更快速的产生中断，除此之外它与 RIQ 中断没什么区别。SVC 是软中断模式，由软件触发，常用于操作系统中，软中断在本手册将会有非常重要的应用。

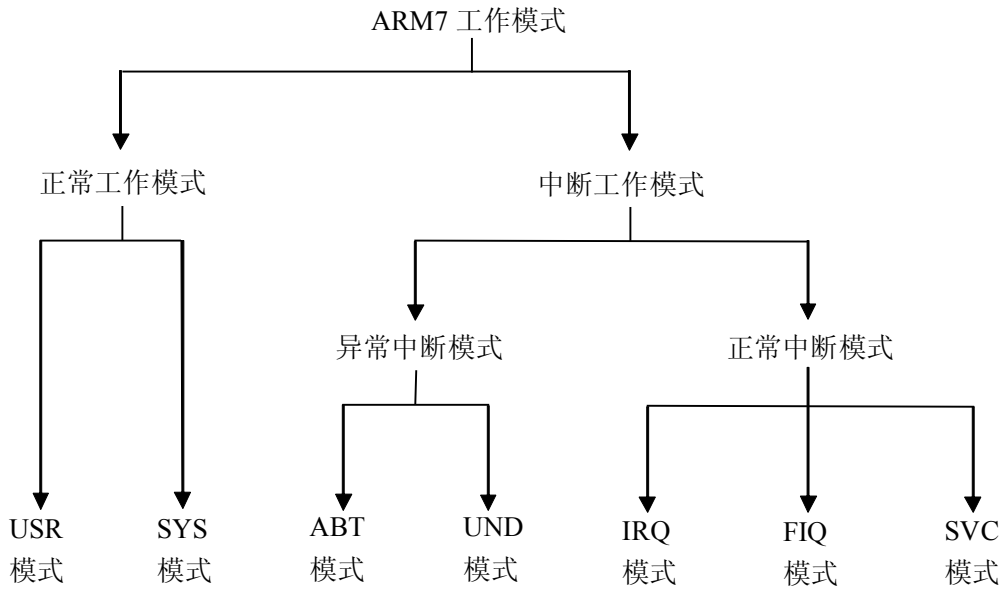


图 2　ARM7 工作模式

程序可以看做是指令+数据的组合，程序运行的过程就是不断的取出指令并按照指令不断计算数据的过程，在这个计算过程中需要使用寄存器来存放指令和数据。寄存器与芯片内核直接相连，因此芯片操作它们的速度要远快于操作内存的速度。但寄存器的数量较少，因此指令和数据平时是存放在 FLASH 或者 RAM 中的，只有当使用时才会放入寄存器进行运算。

每种芯片内部都会有寄存器，不同芯片的寄存器种类、数量各不相同，但都会有下面这 3 种：一、PC（Program Counter）寄存器，PC 寄存器中存放的是当前执行的指令所在的地址，芯片是通过 PC 寄存器找到其需要执行的指令的，更改 PC 寄存器就会发生指令跳转，当我们在 C 语言里调用函数或者产生分支跳转时，实际上就是通过改变 PC 寄存器的值实现的。二、状态寄存器，状态寄存器里都会有 N、Z、C、V 这 4 个状态标志，N 用来表示数据是有符号数还是无符号数，Z 用来表示 0 还是非 0，C 是进位标志，当产生进、借位时影响的就是这个标志，V 是溢出标志，数据运算过程中产生数据溢出了就会更改此标志。三、通用寄存器，这些通用寄存器用来临时存放数据，供芯片运算时使用，某些通用寄存器也可能会有其它专有的功能，各个芯片的定义不一样。

ARM7 芯片每种工作模式下有 17 个寄存器可以使用，分别是 R0~R15 和 CPSR 寄存器，其中 R15 寄存器又可以称之为 PC 寄存器，CPSR 是状态寄存器，其余的可以认为是通用寄存器，在这些通用寄存器里，R13 和 R14 是比较特殊的，R13 寄存器又可以称之为 SP（Stack pointer）寄存器，用来指示当前堆栈的位置，R14 寄存器又可以称之为 LR（Link register）寄存器，当使用某些跳转指令时，硬件会自动将跳转前的指令存入 LR 寄存器中，以供返回时使用。

前面说了 ARM7 芯片有 7 种工作模式，有些寄存器是这 7 种模式共用的，但 ARM7 芯片也为每种不同的工作模式提供了专有的寄存器，进入不同模式便可以使用不同模式下的专有寄存器，如下图所示，ARM7 芯片共有 37 个寄存器，但每种模式仅有 17 个寄存器可以使用。

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

图 3　ARM7 工作模式与寄存器

图 3 中寄存器左下角不带阴影的是不可备份寄存器，各个模式共用，带阴影的是可备份寄存器，为每个模式所单独拥有，只有进入该模式才可以使用。

不可备份寄存器对各个模式来说是共用的，因此，为防止切换后模式破坏切换前模式中的数据，在模式切换后的需要使用软件将这些不可备份寄存器保存起来，当切换回原模式后再恢复这些不可备份寄存器。可备份寄存器则无需软件保存，硬件会在切换模式时自动将切换后模式下的可备份寄存器替换为切换前模式下的可备份寄存器，虽然名字一样，但实际上的物理空间是不同的，因此在不同模式下尽管使用了相同的可备份寄存器，但实际上并没有数据上的冲突，看下面的例子：

在 USR 模式下将 R0 和 R13 的值置为 0：

```
MOV R0, #0
MOV R13, #0
```

此时，发生了从 USR 模式到 IRQ 模式的切换，然后在 IRQ 模式下将 R0 和 R13 加 1：

```
ADD R0, R0, #1
ADD R13, R13, #1
```

R0 是不可备份寄存器，因此上述操作是对同一个 R0 寄存器操作的。在 USR 模式下先将 R0 置为 0，然后在 IRQ 模式下将 R0 加 1，最后 R0 的值为 1。而 R13 是可备份寄存器，在 USR 模式下先将 USR 模式下的 R13 置为 0，进入 IRQ 模式后将 IRQ 模式下的 R13 加 1，最后 USR 模式下的 R13 的值仍为 0，而 IRQ 模式下的 R13 值在它原有的基础上加了 1。在上述操作中，虽然软件使用的都是 R13 这同一个名字，但芯片会根据不同模式而对不同模式下的 R13 寄存器进行操作，上述对 R13 的操作就是对不同的 2 个 R13 寄存器进行的操作。

FIQ 的可备份寄存器是 R8~R14，其它模式的可备份寄存器是 R13~R14，包括 IRQ 模式，因此切换到 FIQ 模式时需要备份的寄存器少一些，因此 FIQ 要比 IRQ 快一些。

CPSR 寄存器不允许被软件随意改写，改写 CPSR 寄存器是需要权限的，只有特权模式，即非 USR 模式才可改写，CPSR 寄存器的结构如下：

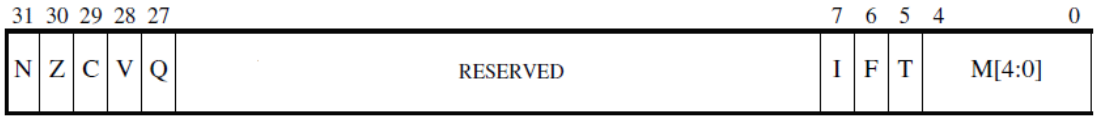


图 4　ARM7 CPSR 寄存器结构

其中 bits27~31 为程序运行时的进位、溢出等标志，前面已经说过一些，具体含义请读者自行查阅附录中的参考文档 2。bits0~4 为芯片工作状态标志，对应关系为：

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARMv4 and above)

图 5　ARM7 芯片模式位

CPSR 与 R 寄存器不同，所有模式下均是使用同一个 CPSP 寄存器，在模式切换时硬件会自动将切换前模式的 CPSR 寄存器保存到切换后模式的 SPSR 寄存器中，然后切换后的模式会继续使用 CPSR 寄存器作为自己的状态寄存器，当需要切换回原有模式时，硬件会自动将 CPSR 从当前模式的 SPSR 寄存器恢复过来。

ARM7 芯片软件的运行完全是由上述的这些寄存器决定的，只要能合理的修改这些寄存器就能控制软件的运行，操作系统就是通过备份、还原、更改这些寄存器来控制程序执行流程的，进而实现任务之间的切换。由于 C 语言无法访问到这些寄存器，因此必须使用汇编语言才能对这些寄存器进行操作，下节我们将了解一些 ARM7 的汇编语言，以便理解操作系统的任务切换过程。

第 2 节 ARM7 汇编语言简介

ARM7 芯片有 2 种汇编语言指令集，一种叫做 ARM 指令集，字长为 32bits，另一种叫 THUMB 指令集，字长为 16bits。这两种指令集各有优缺点，它们可以单独使用也可以混合在一起使用，在 ARM7 芯片上，我们将只使用 ARM 指令集，在后续的 Cortex 芯片上我们将使用 THUMB 指令集的改良版——THUMB2 指令集。

本小节只介绍本操作系统中使用到的一些汇编语言，对它们的介绍也仅限于本操作系统使用到的部分用法，并非全面，更详细的信息请读者自行查阅附录中的参考文档 2。

另外我再补充一下我观点，以前看到一些同学说在学习芯片，请教如何使用汇编语言编程，总是抠这方面的问题。我觉得如果我们学习芯片的目的只是做开发项目，那么就没有必要学习汇编语言，可以把更多的精力放在学习芯片的功能特性上。一个完备的芯片产品甚至不需要底层软件工程师了解太多的芯片硬件外设特性，有封装好的驱动库函数可以直接调用。这次如果不是编写操作系统，我对汇编语言也仅仅是了解一点。汇编语言了解一点即可，，在某些极少数情况下可能会使用到汇编语言定位问题，但这也是极少的情况。

在操作系统中我们使用了下面几条指令：

◆ MOV/MOVS

MOV 是英文单词 Move 的缩写，“搬移”的意思，将数据搬移进寄存器，指令格式为：

```
MOV 目的寄存器, 源寄存器
```

MOV 指令将源寄存器中的数据搬移到目的寄存器中，寄存器间数据搬移可以使用 MOV 指令，如：

```
MOV R0, R1
MOV R14, PC
```

意为：

```
R0 = R1
R14 = PC + 8
```

注意，ARM7 有两级流水线，如果读取 PC 寄存器的话，就会多读取 2 条指令的长度，也就是 8 个字节，目的寄存器为 PC+8。

MOVS 指令与 MOV 指令的格式、功能是一样的，除此之外，如果目的寄存器是 PC 的话，MOVS 会将当前模式下的 SPSR 写入到 CPSR 中。本操作系统从 SVC 模式返回 USR 模式时就需要使用 MOVS 指令恢复 USR 模式的 CPSR。例如，在中断模式下有下面的指令：

```
MOVS PC, R14
```

意为：

```
CPSR = SPSR
```

```
PC = R14
```

◆ ADD

ADD 指令顾名思义，就是英文 Add“加”的意思，指令格式为：

```
ADD 目的寄存器，源寄存器，立即数
```

ADD 指令将源寄存器中的数据和立即数相加的结果保存到目的寄存器中，执行加法操作时可以使用 ADD 指令，如：

```
ADD R14, R14, #0x40
```

意为：

```
R14 = R14 + 0x40
```

◆ SUB/SUBS

SUB 是英文单词 Subtract 的缩写，意为“减”，指令格式为：

```
SUB 目的寄存器，源寄存器，立即数
```

SUB 指令将源寄存器中的数据减去立即数，所得的结果存入到目的寄存器中，执行减法操作时可以使用 SUB 指令，如：

```
SUB R14, R14, #4
```

意为：

```
R14 = R14 - 4
```

SUBS 指令中的 S 标志与 MOVS 指令中的 S 标志作用类似，如果目的寄存器是 PC 的话，SUBS 会将当前模式下的 SPSR 写入到 CPSR 中。本操作系统从 IRQ 中断模式返回 USR 模式时就需要使用 SUBS 指令恢复 USR 模式的 CPSR，如：

```
SUBS PC, R14, #4
```

意为：

```
PC = R14 - 4
CPSR = SPSR
```

◆ AND

AND 指令顾名思义，就是英文 And“与”操作的意思，指令格式为：

```
AND 目的寄存器，源寄存器 1，源寄存器 2
```

AND 指令将源寄存器 1 中的数据和源寄存器 2 中的数据做与操作，结果存入目的寄存器中，执行与操作时可以使用 AND 指令，如：

```
AND R0, R0, R1
```

意为：

```
R0 = R0 & R1
```

◆ ORR

ORR 对应的英文是 Or，“或”操作的意思，指令格式为：

```
ORR 目的寄存器，源寄存器 1，源寄存器 2
```

ORR 指令将源寄存器 1 中的数据和源寄存器 2 中的数据做或操作，结果存入目的寄存器中，执行或操作时可以使用 ORR 指令，如：

```
ORR R0, R0, R1
```

意为：

```
R0 = R0 | R1
```

◆ LDR

LDR 是英文 Load Register 的缩写，“加载寄存器”的意思，将内存中的数据存入寄存器中，指令格式为下面 2 种格式：

```
LDR 目的寄存器，[源寄存器]
LDR 目的寄存器，=常量
```

第一种格式将源寄存器中数据指向的内存地址中的数据存入目的寄存器，第二种格式将常量值存入目的寄存器，为寄存器赋值时可以使用 LDR 指令，如：

```
LDR R14, [R0]
LDR R0, =gpstrCurTaskSpAddr
```

第一条指令意为：

```
R14 = *R0
```

第二条指令中 gpstrCurTaskSpAddr 是一个全局变量，第二条指令意为：

```
R0 = &gpstrCurTaskSpAddr
```

从上述介绍来看，LDR 指令与 MOV 指令似乎具有相同的功能，都可以为寄存器赋值。这两条指令的部分功能确实是一样的，但它们也有各自应用的特点。要说明这两者之间的区别，我们还需要进一步了解 ARM7 的指令结构，以下是有关 ARM7 机器码的一些知识，可以了解一下，但如果你只是做软件开发则不会有太多用处，了解即可。

ARM7 内核采用的是 RISC 精简指令集，所有的 ARM 指令都是 32bits 的，在这 32bits 里既包含了指令的指令码，也包含了指令需要运算的数据，以 MOV 指令为例，通过 MOV 指令的 32bits 可以识别出这是一个 MOV 指令，又可以在这 32bits 里找到源寄存器和目的寄存器。我们来看一下 MOV 指令的机器码格式：

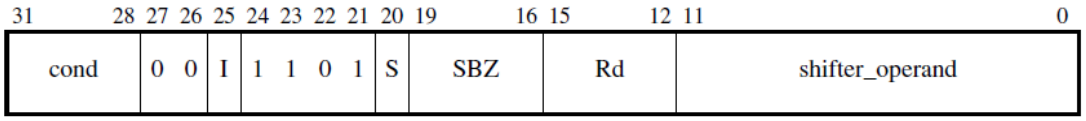


图 6 MOV 指令的机器码格式

28~31bits（cond）是条件码，就是表明这条语句里是否有大于、等于、非零等的条件判断，这 4bits 共有 16 种状态，分别为：

二进制码	指令符号	含义	二进制码	指令符号	含义
0000	EQ	相等	0001	NE	不等
0010	CS/HS	进位/无符号数大于等于	0011	CC/LO	清进位/无符号数小于
0100	MI	减/负数	0101	PL	加/正数或 0

0110	VS	溢出	0111	VC	没溢出
1000	HI	无符号数大于	1001	LS	无符号数小于等于
1010	GE	有符号数大于等于	1011	LT	有符号数小于
1100	GT	有符号数大于	1101	LE	有符号数小于等于
1110	AL	任何条件	1111	-	未定义

表 1　汇编语言条件码

指令与条件码可以有多种组合，比如 MOV 指令可以有 MOV、MOVEQ、MOVLT 等多种形式。前面我们说过状态寄存器里有 NZCV 的状态标志，当执行一条指令时，芯片就会将这条指令的条件码与状态寄存器中的状态标志做比较，如果状态寄存器中的状态标志满足这条指令的条件码时，则执行这条语句，如果不满足则不执行这条指令。状态寄存器中的状态标志是受某些指令影响的，因此在使用有条件码的指令进行判断前，必然会有其它指令配合使用，先修改状态寄存器中的状态标志，例如：

```
CMP    R1, #0
BEQ    GETNEXTTASKSP
```

第一条指令“CMP”是一个“比较”指令，如果 R1 等于 0，那么它就将状态寄存器中的 Z 置为 1，表示结果为真，否则，将状态寄存器中的 Z 置为 0，表示结果为假。第二条指令其实是一条“B”指令，是“跳转”指令，B 之后的“EQ”就是条件码，从表 1 中可以知道，条件是“相等”时才执行。

当 R1 等于 0 时，CMP 指令就将 Z 置为 1，执行 BEQ 时满足条件，就执行了跳转。如果 R1 不等于 0，CMP 指令就将 Z 置为 0，执行 BEQ 时不满足条件，就不执行跳转。

同理，只有当状态寄存器中的标志为相等时，MOVEQ 指令才会执行，这时其功能与 MOV 指令相同。而 MOVLT 指令则是当状态寄存器中的标志为有符号数，并且处于小于状态时才会执行的 MOV 指令。MOV 指令的条件码是 AL，因此 MOV 指令可以不管任何条件都去执行。其它指令也可以与条件码组合使用，具体情况请查阅参附录中的参考文档 2。

25bit（I）是用来区别 shifer_operand 域是采用立即数寻址方式还是寄存器寻址方式，该 bit 为 0 表示寄存器寻址方式，为 1 表示立即数寻址方式，这就涉及到了指令的寻址方式。

寻址方式的出现不是为了使指令能有多种写法，而是受指令长度限制被迫产生的产物。以 MOV 指令为例，如果采用立即数寻址，立即数的长度不可能超过 shifer_operand 域的长度（MOV 指令可以采用移位的方式装下部分更长的立即数，这些不在讨论之内），因此我们就不可能使用

```
MOV R0, #0x12345678
```

这条指令。立即数#0x12345678 是 32bits 数据，已经超过了 shifer_operand 域所能装下的最长 12bits 数据，如果把 0x12345678 全部被存到指令中，那么该指令中将无法存储条件码等其它指令信息，因此，这条指令在编译时就会报错。

为了解决这个问题，芯片设计人员就设计了寄存器寻址方式，在 ARM7 中每种模式有 16 个通用寄存器，2 的 4 次方等于 16，因此只需要用 4bits 就可以为每个寄存器分配一个编号，R0~R15 寄存器分别对应 0~15 的编号。4bits 的寄存器编号完全可以存入 shifer_operand 域。采用寄存器寻址时，指令先查到寄存器的编号，然后再从寄存器中取出使用的数据，这样就解决了 MOV 指令受指令长度的限制而无法操作长立即数的问题。

从上述描述的过程来看采用寄存器寻址方式必须先将数据放入一个寄存器中，然后才能使用 MOV 指令采用寄存器寻址。对比立即数寻址方式，它增加了指令的执行时间，也增加了代码，还多用了一个寄存器，但它的优点是可以操作长的数据。

除了上面这两种寻址方式外，ARM7 还有多种其它寻址方式，每种寻址方式都有其自身的特点，适用不同的场景，这里不介绍了。

21~24bits（opcode）是指令码，用来表明这条指令是什么指令，例如，MOV 指令的指令码是 0b1101，看到 0b1101，芯片就将这条指令当做 MOV 指令来解析。

20bit（S）就是指令中 S 标志的体现，该 bits 为 0 表示指令不带 S，为 1 表示指令带 S，功能见上述指令介绍。

16~19bits（SBZ）手册中没查到是什么意思，SBZ 应该是 should be zero 的意思，对比了几条指令发现该域果然全是 0，应该是保留位。

12~15bits（Rd）是指令中的目的寄存器，存放寄存器的 4bits 编号。

0~11bits（shifter_operand），指令的操作数。

下面我找了 4 条指令，将 MOV 指令做一个对比：

指令	机器码	指令格式							
		cond	00	I	opcode	S	SBZ	Rd	shifer_operand
MOV R1, #0x64	E3A01064	1110	00	1	1101	0	0000	0001	000001100100
		条件码为 1110 适用任何条件		立即数方式	MOV 的指令码	指令没有 S 标志		目的寄存器为 R1	源操作数为立即数 0x64
MOVS PC, R14	E1B0F00E	1110	00	0	1101	1	0000	1111	000000001110
		条件码为 1110 适用任何条件		寄存器方式	MOV 的指令码	指令有 S 标志		目的寄存器为 R15	源操作数为寄存器 R14
MOVLT R3, #0x1	B3A03001	1011	00	1	1101	0	0000	0011	000000000001
		LT 的条件码为 1011		立即数方式	MOV 的指令码	指令没有 S 标志		目的寄存器为 R3	源操作数为立即数 1
MOVEQ R0, R1	01A00001	0000	00	0	1101	0	0000	0000	000000000001
		EQ 的条件码为 0000		寄存器方式	MOV 的指令码	指令没有 S 标志		目的寄存器为 R0	源操作数为寄存器 R1

表 2 MOV 指令汇编格式对比

LDR 指令可以将 32bits 数据一次装入寄存器中，这里不再详细说明了，请读者自行参考文档。

◆ STR

STR 是英文 Store Register 的缩写，“存储寄存器”的意思，将数据从寄存器存入内存。STR 指令与 LDR 指令功能相反，指令格式为：

```
STR 源寄存器,[目的寄存器]
```

STR 指令将源寄存器中的数据存入目的寄存器中数据所指向的内存地址，将寄存器中的数据存入内存时可以使用 STR 指令，如：

```
STR R1,[R0]
```

意为：

```
*R0 = R1
```

◆ LDM

LDM 对应的英文是 Load Multiple，LDM 指令是 LDR 指令的增强版，可以将多个连续的内存数据存入一组寄存器中，这条指令在堆栈操作中经常使用，在介绍这条指令前我们先了解一下堆栈。

堆栈是分配在内存中的一部分空间，但堆和栈是 2 个概念，用户调用 malloc 等函数申请的内存就是从堆中申请的，这块内存使用完需要由用户自行释放，堆是由用户管理的。当发生函数调用时，程序会自动将父函数的寄存器存入内存，这部分内存就叫做栈，当子函数返回父函数时，程序会从栈中取出保存的寄存器数值，再恢复到寄存器中，这样就完成了一次函数调用，栈是由程序自动管理的。

栈有空满之分，栈有增减之分。根据栈指针不同的操作方式，可以将栈分为 4 种。栈指针指向栈顶的元素，即最后一个入栈的元素，此时栈指针指向的栈空间是用过的，是满的，这种栈叫做满（Full）栈。栈指针指向与栈顶元素相邻的下一个元素，此时栈指针指向的栈空间是没用过的，是空的，这种栈叫做空(Empty)栈。当向栈存储数据时，栈指针是向着内存地址减少的方向移动的，这种栈叫做递减（Descending）栈。当向栈存储数据时，栈指针是向着内存地址增长的方向移动的，这种栈叫做递增（Ascending）栈。综合栈的空满和增减特性，栈可以分为 FD ED FA EA 这 4 种类型，我们所使用的 ARM7 芯片是 FD 类型。

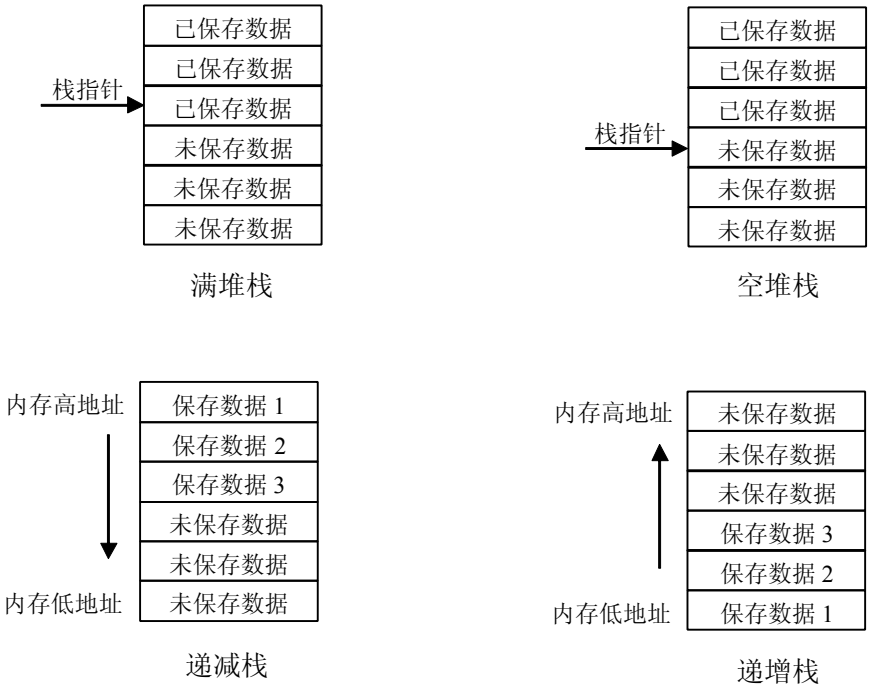


图 7　栈的 4 种类型

尽管 ARM7 芯片是 FD 类型，但这并不意味着栈指针必须指向栈顶，也不意味着存入数据时栈指针必须移向内存低端，因为汇编指令里提供了 4 种对栈的操作方式，分别是 DB（Decrement Before）、DA（Decrement After)）、IB（Increment Before）和 IA（Increment After）。DB 意为栈指针先减少然后再操作，DA 意为先操作然后栈指针再减少，IB 意为栈指针先增加然后再操作，IA 意为先操作然后栈指针再增加。这 4 种操作方式都可以与 LDM 指令组合，

形成 LDMDB、LDMDA、LDMIB 和 LDMIA 指令。

有了上述 4 种 LDM 指令我们就可以对堆栈灵活的操作了，LDM 有下面 3 种指令格式（不考虑 LDM 的后缀）：

```
LDMIA 源寄存器,{一组目的寄存器}
LDMIA 源寄存器!,{一组目的寄存器}
LDMIB 源寄存器,{一组目的寄存器}^
```

第一种指令格式从源寄存器指定的内存地址开始，将一组数据从内存存入到这组目的寄存器中，从内存取数据的方向由 LDM 指令后缀来确定。目的寄存器之间可用“,”分开，也可用“-”表示一个范围的寄存器，具体见下面例子。第二种指令格式除了完成第一种指令格式的功能外，还将操作后的源寄存器数值保存到源寄存器中。第三种指令格式不能在正常工作模式下使用，目的寄存器中也不能包含 PC 寄存器，这种指令格式是在中断工作模式下，访问 USR 模式下的目的寄存器，而不是当前模式下的寄存器，看下面例子：

```
LDMIA R13,{R0 - R5}
LDMIA R13!,{R0 - R3, R12}
LDMIB R14,{R10 - R14}^
```

第一条指令从 R13 寄存器指向的内存地址连续取出 6 个 32bits 数据存入到 R0~R5 寄存器中，意为：

```
R5 = *R13
R4 = *(R13 + 4)
R3 = *(R13 + 8)
R2 = *(R13 + 12)
R1 = *(R13 + 16)
R0 = *(R13 + 20)
```

操作完之后，R13 的数值不变，仍为操作前的栈地址。

第二条指令从 R13 寄存器指向的内存地址连续取出 5 个 32bits 数据存入到 R0~R3 和 R12 寄存器中，并且保存操作后的 R13 数值，意为：

```
R12 = *R13
R3 = *(R13 + 4)
R2 = *(R13 + 8)
R1 = *(R13 + 12)
R0 = *(R13 + 16)
R13 = R13 + 20
```

操作完成后，将 R13 更新为操作后的栈地址。

假设当前为 IRQ 模式，第三条指令从 IRQ 模式的 R14 寄存器指向的内存地址连续取出 5 个 32bits 数据存入 R10~R12 寄存器和 USR 模式的 R13、R14 寄存器中，意为：

$R14_{USR}$ = *($R14_{IRQ}$ + 4)
$R13_{USR}$ = *($R14_{IRQ}$ + 8)
R12 = *($R14_{IRQ}$ + 12)
R11 = *($R14_{IRQ}$ + 16)
R10 = *(R14IRQ + 20)

下面我们再看一个例子，看看 LDM 指令使用不同的后缀会有什么不同。

```
LDMDB R13,{R0 - R3}
```

等同于

```
R3 = *(R13 - 4)
```

```
R2 = *(R13 - 8)
R1 = *(R13 - 12)
R0 = *(R13 - 16)
```

```
LDMDA R13, {R0 - R3}
```

等同于

```
R3 = *(R13)
R2 = *(R13 - 4)
R1 = *(R13 - 8)
R0 = *(R13 - 12)
```

```
LDDIB R13, {R0 - R3}
```

等同于

```
R3 = *(R13 + 4)
R2 = *(R13 + 8)
R1 = *(R13 + 12)
R0 = *(R13 + 16)
```

```
LDMIA R13, {R0 - R3}
```

等同于

```
R3 = *(R13)
R2 = *(R13 + 4)
R1 = *(R13 + 8)
R0 = *(R13 + 12)
```

这 4 种形式分别与 C 语言中断的--i、i--、++i、i++类似。

◆ STM

STM 对应的英文是 Store Multiple，STM 指令是 STR 指令的增强版，可以将一组寄存器中的数据存入到内存中。同样，STM 也有 4 种操作方式，STMDB、STMDA、STMIB 和 STMIA，分别与 LDM 对应。

指令格式为（不考虑 STM 的后缀）：

```
STMIA 目的寄存器, {一组源寄存器}
STMDB 目的寄存器!, {一组源寄存器}
STMIA 目的寄存器, {一组源寄存器}^
```

第一种指令格式将一组源寄存器内的数据存入连续的目的寄存器所指向的内存空间，存入内存的数据方向由 STM 指令的后缀来确定，源寄存器之间可用“,”分开，也可用“-”表示一个范围的寄存器，具体见下面例子。第二种指令格式除完成第一种指令格式的功能外，还将操作后的目的寄存器数值保存到目的寄存器中。第三种指令格式不能在正常工作模式下使用，源寄存器中也不能包含 PC，这种指令格式中所访问的源寄存器是 USR 模式下的寄存器，而不是当前模式下的寄存器，看下面例子：

```
STMIA R14, {R0}
STMDB R13!, {R0 - R3, R12, R14}
```

```
STMIA R14, {R10 - R14}^
```

第一条指令将 R0 存入 R14 指向的内存，意为：

```
*R14 = R0
```

操作完之后，R14 的数值不变，仍为操作前的栈地址。

第二条指令将 R0~R3、R12 和 R14 寄存器中的数据存入从 R13 开始的地址，并且保存操作后的 R13 数值，意为：

```
*(R13 - 4) = R14
*(R13 - 8) = R12
*(R13 - 12) = R3
*(R13 - 16) = R2
*(R13 - 20) = R1
*(R13 - 24) = R0
R13 = R13 - 24
```

操作完成后，将 R13 更新为操作后的栈地址。

假设当前为 IRQ 模式，第三条指令将 IRQ 模式的 R10~R12 和 USR 模式的 R13~R14 寄存器中的数据存入到 IRQ 模式的 R14 寄存器指向的地址，意为：

$*R14_{IRQ} = R14_{USR}$
$*(R14_{IRQ} + 4) = R13_{USR}$
$*(R14_{IRQ} + 8) = R12$
$*(R14_{IRQ} + 12) = R11$
$*(R14_{IRQ} + 16) = R10$

◆ BL

BL 是英文 Branch and Link 的缩写，“跳转并连接”的意思，BL 指令会在跳转到目的地址的同时将 BL 指令的下条指令地址存入 LR 寄存器中，指令格式为：

```
BL 目的地址
```

BL 指令跳转到目的地址，同时将 BL 指令的下条指令地址存入 LR 寄存器供程序返回时使用，调用函数时可以使用 BL 指令，如：

```
0x00080398 EB0002E2  BL     0x00080F28
```

0x00080398 是 BL 指令所在的地址，EB0002E2 是 BL 指令的机器码，0x00080F28 是 BL 指令要跳转到的地址。从这个格式来看，BL 指令好像是一个绝对跳转指令，直接跳转到 0x00080F28 这个地址，其实不然，BL 指令是一个相对跳转指令，格式如下：

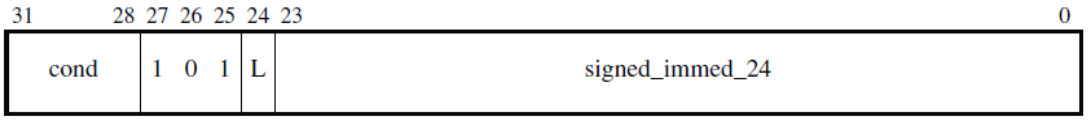


图 8 B 指令的机器码格式

BL 指令的 0~23bits 存放的是要跳转的相对地址，由于指令所在地址必须是 4 字节对齐的，因此跳转的地址最低 2bits 必然是 0，因此 BL 指令 0~23bits 保存的是省略这最低 2bits 的地址，如果补全了这 2bits，BL 指令就可以表示 26bits 的跳转地址。在这 26bits 中需要使用 1bit 表示向前跳还是向后跳，那么剩下的 25bits 就可以表示 32MByts 的范围了，2^{25}=32M，因此，我们在很多文档上可以看到 B 跳转指令只能跳转到±32MBytes 范围内的说明，就是这个原因。

上面这个 BL 指令要跳转的相对地址是 0x2E2（BL 指令 0~23bits），补充 2 个最低位后，跳转的相对地址为 0xB88，由于 ARM7 有 2 级流水线，所以跳转到的指令需要多加 8 个字节，BL 要跳转的实际地址为 0x00080398+0xB88+8=0x00080F28。

这条 BL 指令执行下面的操作：

```
LR = 0x0008039C
PC = 0x00080F28
```

在操作系统中我们没有使用 BL 指令，就是因为我们不知道我们所调用的函数是否会超出 BL 指令的跳转范围，但我们可以看到编译器编译出的很多程序都是使用 BL 指令调用函数的，编译器之所以不怕跳转超出±32MBytes 的范围，是因为编译器在编译时就知道了程序所需要跳转的范围，它会为±32MBytes 之内的跳转分配 BL 指令，保证 BL 指令不会超出范围。在这里以 BL 指令为例，介绍一下 B 指令的相关知识。

◆ BX

BX 是英文 Branch and Exchange 的缩写，“跳转并改变状态”的意思，BX 指令除了可以实现跳转，还可以改变芯片运行的指令集，可以在 ARM 指令集与 THUMB 指令集之间切换，这里我们只使用了它的跳转功能，格式为：

```
BX 目的寄存器
```

BX 指令跳转到目的寄存器中存储的地址，由于寄存器可以存放 32bits 数据，因此 BX 指令可以实现芯片全空间跳转。

◆ MRS

MRS 是英文 Move PSR to general-purpose register 的缩写，“将 PSR 寄存器的内容保存到通用寄存器中”的意思，就是将 CPSR 或 SPSR 寄存器的内容保存到 R 寄存器中，格式为：

```
MRS R0, SPSR
```

意为：

```
R0 = SPSR
```

◆ MSR

MSR 是英文 Move general-purpose register to PSR 的缩写，“将通用寄存器的内容保存到 PSR 寄存器中”的意思，就是将 R 寄存器的内容保存到 CPSR 或 SPSR 寄存器中，格式为：

```
MSR SPSR, R0
```

意为：

```
SPSR = R0
```

◆ NOP

NOP 是 NO Operation 的缩写，意为空指令，执行该指令时芯片什么也不做，空闲一个指令周期。

在 ARM7 芯片上，我们有了上述的汇编知识就足够编写操作系统了。

指令先介绍到这里，我们再来看看汇编的函数如何来写。汇编函数使用“.func”作为函

数的开始，使用“.end”标志着函数的结束，例如

```
    .func TaskOccurSwi
TaskOccurSwi:
    SWI
    BX      R14
    .endfunc
```

这就是用汇编语言写的一个函数——TaskOccurSwi。

在这里我们使用的是 GNU 编译器，在 GNU 的编译器中“@”代表注释符，与 C 语言中的//是一样的效果，它之后的所有语句都被认为是注释，例如：

```
@CMP    R1, #0
```

这条语句不能执行，编译器根本就不会将它编译进来。

第 3 节 ARM7 芯片的函数调用标准

在上节，我们最后用汇编语言写了一个函数，但该函数没有入口参数，那么 C 语言函数、汇编函数之间是如何传递参数和返回值的？函数在执行过程中是如何使用栈的？它们需要遵守什么规则？本节我们将介绍这方面的内容。

如果我们不是在编写操作系统，只是编写正常的 C 函数，那么我们是不需要关心函数调用的细节，编译器会遵守一定的函数调用规则编译成二进制代码，当所有不同类型的编译器都遵守这个相同的规则时，各种编译器编译出来的程序就可以互相配合运行了。这个规则就是 AAPCS——Procedure Call Standard for the ARM Architecture，即附录中的参考文档 3。如今，我们编写操作系统需要改变 C 函数标准的运行方式，但我们仍必须遵守这个规则，这样才能与编译器编译出来的代码配合使用。

AAPCS 对 ARM 结构的一些标准做了定义，在这里我们只重点介绍函数调用部分，如图 9 所示，AAPCS 为 ARM 的 R0~R15 寄存器做了定义，明确了它们在函数中的职责：

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

图 9　AAPCS 关于 ARM 寄存器的定义

函数调用时的规则如下：

1. 父函数与子函数间的入口参数依次通过 R0~R3 这 4 个寄存器传递。父函数在调用子函数前先将参数存入到 R0~R3 中，若只有一个参数则使用 R0 传递，2 个则使用 R0 和 R1 传递，依次类推，当超过 4 个参数时，其它参数通过栈传递。当子函数运行时，根据自身参数个数自动从 R0~R3 或者栈中读取参数。
2. 子函数通过 R0 寄存器将返回值传递给父函数。子函数返回时，将返回值存入 R0，当返回到父函数时，父函数读取 R0 获得返回值。
3. 发生函数调用时，R0~R3 是传递参数的寄存器，即使是父函数没有参数需要传递，子函数也可以任意更改 R0~R3 寄存器，无需考虑会破坏它们在父函数中保存的数值，返回父函数前无需恢复其值。AAPCS 规定，发生函数调用前，由父函数将 R0~R3 中有用的数据压栈，然后才能调用子函数，以防止父函数 R0~R3 中的有用数据被子函数破坏。
4. R4~R11 为普通的通用寄存器，若子函数需要使用这些寄存器，则需要将这些寄存器先压栈然后再使用，以免破坏了这些寄存器中保存的父函数的数值，子函数返回父函数前需要先出栈恢复其数值，然后再返回父函数。AAPCS 规定，发生函数调用时，父函数无需对这些寄存器进行压栈处理，若子函数需要使用这些寄存器，则由子函数负责压栈，以防止父函数 R4~R11 中的数据被破坏。
5. 编译器在编译时就确定了函数间的调用关系，它会使函数间的调用遵守 3、4 条规定。但编译器无法预知中断函数的调用，被中断的函数无法提前对 R0~R3 进行压栈处理，因此需要在中断函数里对它所使用的 R0~R11 压栈。对于中断函数，不遵守第 3 条规定，遵守第 5 条规定。
6. R12 寄存器在某些版本的编译器下另有它用，用户程序不能使用，因此我们在编写汇编函数时也必须对它进行压栈处理，确保它的数值不能被破坏。

7. R13 寄存器是堆栈寄存器（SP），用来保存堆栈的当前指针。
8. R14 寄存器是链接寄存器（LR），用来保存函数的返回地址。
9. R15 寄存器是程序寄存器（PC），指向程序当前的地址。

上述只介绍了本手册中使用到的情形，具体的情况在编写操作系统代码时会涉及到，其它规则请请读者自行查找资料。

接下来我们再通过几个小例子熟悉一下 C 函数与汇编函数的调用过程。下面的 C 函数 TestFunc1 与汇编函数 TestFunc2 的功能是一样的。

```
U8 TestFunc1(void)
{
    U8 ucPara1;
    U8 ucPara2;
    U8 ucPara3;
    U8 ucPara4;
    U8 ucPara5;
    U8 ucPara6;

    ucPara1 = 1;
    ucPara2 = 2;
    ucPara3 = 3;
    ucPara4 = 4;
    ucPara5 = 5;
    ucPara6 = 6;

    return ucPara1 + ucPara2 + ucPara3 + ucPara4 + ucPara5 + ucPara6;
}
```

```
    .func TestFunc2
TestFunc2:

    STMDB R13!, {R5 - R6, R10}    @R5，R6，R10 寄存器压栈
    LDR R1, =1
    LDR R3, =2
    LDR R4, =3
    LDR R5, =4
    LDR R6, =5
    LDR R10, =6

    ADD R0, R1, R3
    ADD R0, R0, R4
    ADD R0, R0, R5
    ADD R0, R0, R6
    ADD R0, R0, R10

    LDMIA R13!, {R5 - R6, R10}    @R5，R6，R10 寄存器出栈

    .endfunc
```

TestFunc2 函数使用了 R0、R1、R3、R4、R5、R6、R10 共 7 个寄存器，遵循 AAPCS 规则，在使用 R0、R1 和 R3 之前并没有对它们压栈，但对 R5、R6 和 R10 寄存器进行了压栈保存，在函数返回前又出栈还原了这 3 个寄存器，这样 TestFunc2 函数返回到它的父函数之后，R5、R6 和 R10 寄存器的数值是没有改变的，而 R0、R1、R3 和 R4 则分别被改写为了 21、1、2 和 3。

下面我们再来看看 C 函数 TestFunc3 调用汇编函数 TestFunc4 完成 1+2 的运算。

```
U8 TestFunc3(void)
{
    return TestFunc4(1, 2);
}


    .func TestFunc4
TestFunc4:

    ADD R0, R0, R1
    BX R14;

    .endfunc
```

TestFunc3 函数在调用 TestFunc4 函数前已经将参数 1 和 2 分别存入 R0 和 R1，并将返回地址存入到 R14 中，然后才跳转到 TestFunc4 函数，发生函数调用。这时程序将运行 TestFunc4 函数，它将 R0 和 R1 相加，将结果放入 R0，需要通过 R0 将返回值返回给 TestFunc3 函数。此时 R14 中保存的就是返回 TestFunc3 函数的返回地址，最后 TestFunc4 函数跳转到 R14 就返回到了 TestFunc3 函数，TestFunc3 函数从 R0 就可以取出 TestFunc4 函数计算的结果了。

下面我们再来看看汇编函数 TestFunc5 调用 C 函数 TestFunc6 完成 1+2 的运算。

```
    .func TestFunc5
TestFunc5:

    MOV R0, #1
    MOV R1, #2
    SUB R13, R13, #4
    STR R14, [R13]
    BL TestFunc6
    LDR R14, [R13]
    ADD R13, R13, #4
    BX R14

    .endfunc


U8 TestFunc6(U8 ucPara1, U8 ucpara2)
{
    return ucPara1 + ucPara2;
}
```

TestFunc5 函数先将参数 1 和 2 存入 R0 和 R1 寄存器，准备调用 TestFunc6 函数并传递入口参数，然后将 R14 寄存器压栈，以防止使用 BL 指令时存入的 R14 返回地址破坏 R14 原有的数据，然后调用 TestFunc6 函数。在调用 TestFunc6 函数时 BL 指令会自动将“LDR R14, [R13]”这条指令的地址存入 R14，这样就开始运行 TestFunc6 函数了。TestFunc6 函数会自动从 R0 和 R1 寄存器中取出参数，将计算结果存入 R0，通过 R0 将返回值返回给 TestFunc5 函数。TestFunc6 函数跳转回 TestFunc5 函数后，TestFunc5 函数从栈中恢复原有的 R14 寄存器，完成函数调用，此时 R0 中的数值就是 TestFunc6 函数的计算结果。

当函数比较简单，不需要压栈仅使用寄存器便可以完成运算的时候，那么下面的 TestFunc7 函数，它的返回值是多少？

```
U8* TestFunc7(void)
{
    U8 ucPara1;

    ucPara1 = 1;

    return &ucPara1;
}
```

按照上面的分析，对于这个简单的函数，编译器是不会为局部变量 ucPara1 分配内存空间的，ucPara1 只会保存在寄存器中，因此无从谈起它的地址。但这个这么简单的函数却偏偏要获取这个仅在寄存器中的局部变量的地址，遇到这种情况，编译器在编译时会特别为 ucPara1 专门在栈中分配内存，因此也就可以获取到它的地址了。

当然，这个函数没有任何意义，仅是举一个例子，而且写 C 语言时要避免发生这种情况，因为 TestFunc7 函数返回的是栈内局部变量的地址，当 TestFunc7 函数运行完后，ucPara1 这个局部变量所在的栈空间已经被释放，这个栈空间很可能已经被其它变量占用，如果这时候还使用这个地址的话就可能会导致系统崩溃，新手要避免产生这个错误。

第 4 节 Wanlix 的文件组织结构

说起写软件，还是比较容易入门的，现在电脑这么普及，随便找本软件的书籍就可以在电脑上编程了，实现一些功能，但这仅仅是编写软件的最初级阶段，一部分人可能一辈子只会停留在这个阶段，全局变量满天飞，函数没有层次结构，文件关系混乱。能够发展下去，能够编出满足功能需求，可维护性、可测试性好，效率高，用户易用的软件才可称之为软件人员。编码只是软件中很小的一个环节，随着产品不断的扩大，这一点越来越明显，编码固然重要，但编码之外的设计也非常重要。

我写的代码虽有一些条理，但也比较凌乱，还请各位多多包涵，就算是一个反面教材，同时，也希望大家能写出好的软件！

现在虽然是在写操作系统，但操作系统最终是要给用户使用的，为了方便用户使用，我们需要设计一下文件结构。如图 10 所示，RTOS_Wanlix 是整个项目的根目录，下面包含了 wanlix、srccode、others、outfile 和 project 这 5 个目录。与操作系统相关的文件被放在 wanlix 目录下。用户文件用来实现产品功能，放在 srccode 目录下。编译后的输出文件放在 outfile 目录下。我使用的是 Keil 开发工具，与 Keil 相关的工程文件放在 project 目录下。其它文件放在 others 目录下。

```
RTOS_Wanlix
 ├[wanlix]
 │    ├[wanlix.h]
 │    ├[wlx_core_a.asm]
 │    ├[wlx_core_a.h]
 │    ├[wlx_core_c.c]
 │    └[wlx_core_c.h]
 ├[srccode]
 │   ├[global.h]
```

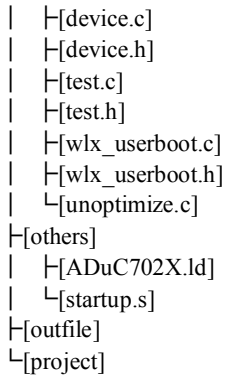


图 10 Wanlix 文件结构

下面详细介绍各个目录和文件。

- wanlix 目录中存放的是操作系统的源文件，所有的操作系统文件均是以“wlx_”为前缀，操作系统头文件 wanlix.h 除外。
  - ✓ wanlix.h 文件是操作系统的总头文件，定义了操作系统共用的宏、结构体，供操作系统全部文件使用，也是操作系统对外的接口文件。用户代码只需要包含且仅需要包含这个头文件，就可以使用 Wanlix 操作系统的所有功能了。
  - ✓ wlx_core_a.asm 文件是使用汇编语言编写的操作系统内核调度文件，所有与汇编相关的代码都放在这个文件里。
  - ✓ wlx_core_a.h 文件是 wlx_core_a.asm 文件的头文件，被 wlx_core_a.asm 文件包含，定义了 wlx_core_a.asm 文件使用的宏、声明了 wlx_core_a.asm 文件使用的全局变量和函数等。
  - ✓ wlx_core_c.c 文件是使用 C 程序编写的操作系统内核调度文件，这个文件是操作系统的核心文件，与操作系统调度相关的功能都是在这个文件实现的。
  - ✓ wlx_core_c.h 文件是 wlx_core_c.c 文件的头文件，被 wlx_core_c.c 文件包含，定义了 wlx_core_c.c 文件使用的宏、声明了 wlx_core_c.c 文件使用的全局变量和函数等。
- srccode 是用户代码目录，该目录中保存的是用户源代码文件。srccode 目录下的文件是与项目直接相关的，用户可根据自身需要增减、修改文件，可以自行安排。在本手册中使用这些用户文件编写一些例子，用来演示操作系统的功能。
  - ✓ global.h 文件是用户文件的总头文件，用户文件共同使用的信息被存放到该头文件里，该文件被各个用户 c 文件的 h 头文件包含，以便每个用户文件都可以使用共有的接口功能。该头文件包含了 wanlix.h 文件，以便所有用户文件可以使用 Wanlix 的功能。
  - ✓ device.c 文件是驱动文件，设备所有的驱动程序均放在此文件。
  - ✓ test.c 文件包含了演示操作系统功能所使用的代码。
  - ✓ wlx_userboot.c 文件是操作系统与用户代码的接口文件，用户代码从该文件启动。C 语言的入口函数是 main 函数，在 Wanlix 操作系统里 main 函数将被封装到操作系统内部，用户不可见，用户代码将从该文件里的 WLX_RootTask 函数启动。用户需要根据自身需要向该文件添加代码，这也是这个文件放在 srccode 目录的原因。
  - ✓ unoptimize.c 文件里包含的是不能被优化的代码，因此单独提出对该文件采用不优化的编译选项，其它文件均采用 O2 的优化选项。
  - ✓ xxx.h 文件是 xxx.c 文件的头文件，仅包含 xxx.c 文件所使用的信息。

- others 目录里保存的是与开发工具相关的文件，本手册使用的是 Keil 开发工具，这个目录里保存的是 Keil 中所使用的与芯片相关的文件，包括芯片启动文件 startup.s 和链接文件 ADuC702X.ld。
  - ✓ 在 startup.s 文件中包括了芯片的中断向量表以及芯片启动程序，由汇编语言编写。
  - ✓ ADuC702X.ld 文件是整个工程的链接文件，决定了芯片存储空间的分配。
- project 目录是开发工具的文件所在目录，我们使用的是 Keil，所有与 Keil 工程相关的文件均保存在此目录。这个目录里的文件我们不用关心，由 Keil 自动生成。
- outfile 目录是输出文件目录，代码编译后输出的所有文件就存放在这个目录里。

为方便理解这些文件之间的调用关系，我们通过图 11 来做一个说明：

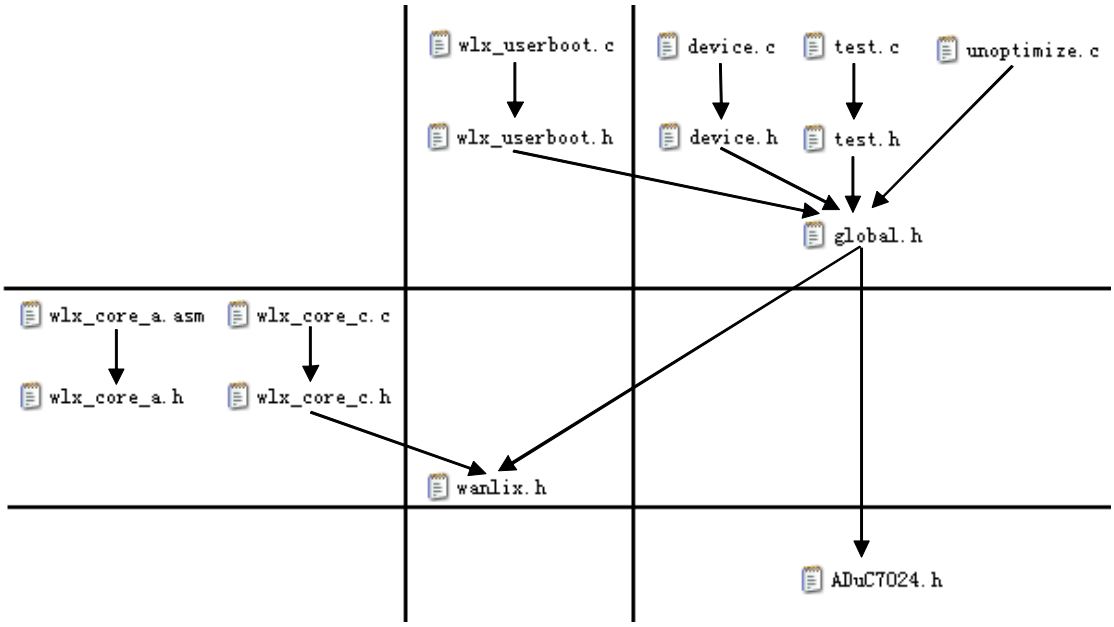


图 11 Wanlix 文件调用关系

顺着箭头的方向代表“包含”的意思，A—>B 表示 A 文件包含 B 文件。

图 11 中最上面一行文件是需要用户自己编写的文件，需要用户自行修改。中间一行是操作系统的文件，用户不能修改。最下面一行是芯片定义的头文件，由芯片厂商提供，用户不能修改。左边一列是操作系统文件，中间一列是操作系统与用户的接口文件，右边一列是用户文件。

其中 mds_core_a. asm 文件有点特殊，因为它是汇编文件，无法使用 C 文件中的定义，因此它与 C 文件没有关系，它里面的函数是放在 mds_core_c.h 文件中声明的。

经过对文件结构的设计，每个 c 文件只需要包含它对应的 h 文件，每个 c 文件的 h 文件都需要包含总头文件，用户文件需要包含 wanlix.h 文件，形成一个树状结构。

第 5 节 Wanlix 的开发环境

芯片使用的是 ADI 公司的 Aduc7024，前面已经做过一些介绍。

软件开发环境使用的是 Keil MDK4.20。Keil 是德国软件公司 Keil（现已被 ARM 公司收

购）开发的嵌入式系统开发平台，Keil 开发平台支持许多厂家的芯片，提供基本的最小软件系统，Keil 开发环境集成了文本编辑器、C 编译器、汇编编译器、链接器等工具，并提供仿真调试功能，可使用仿真器在线硬仿真，也可单独使用 Keil 进行软仿真，仿真时有多种调试手段可以使用。因此，Wanlix 和 Mindows 选择在 Keil 工具下开发的。我所使用的是 MDK4.20 免费版本，有 32KBytes 程序空间的使用限制。

Keil 允许更改其编译工具链，在开发 Wanlix 时，我选择了功能强大的 GNU 编译工具链。本章第 3 节所介绍的汇编语言就是 GNU 中的 ARM7 汇编语言，与其它工具链的汇编语言会有少许出入。

编译选项使用的是 O2 优化，只有 unoptimize.c 文件采用的是 O0 优化。

有关 Keil 开发环境的设置参考附录 3。

第 3 章 Wanlix 操作系统

有了前面章节的铺垫，本章开始正式编写操作系统！本章将实现 Wanlix 操作系统，从零起步，先实现 2 个固定任务的互相切换来验证操作系统的切换功能，然后再不断的加入新功能，由浅入深，一步步将操作系统充实起来。每一个功能的加入都是一个独立的阶段性，读者可以通过附带的视频和图片看到各个阶段的成果。

Wanlix 只提供主动切换任务的功能，是非抢占操作系统，编写相对简单，作为学习编写操作系统的入门教材是个不错的选择。这也使得它非常小巧，适合在硬件资源少但又需要任务切换的小型嵌入式软件系统中使用。

第 1 节 两个固定任务之间的切换

程序的执行只与指令和数据相关，指令是不可修改的，编译后就确定了，能改变的只有数据，但指令需要对数据进行判断，走不同的指令分支，因此，如果我们需要控制程序的执行过程，不但需要编写出指令，还需要提供可方便使用的数据，操作系统任务切换的过程就是指令备份、恢复数据的过程。

通过前面的介绍，我们知道程序当前指令执行的结果只与 R0~R15、CPSR 这 17 个寄存器有关，只要我们能控制这 17 个寄存器，那么我们就可以控制程序的执行流程，这是实现任务切换的基础。

从 C 语言的角度来看，任务就是函数，只不过是在操作系统里，一个任务可以切换到其它任务，其实也就是一个函数可以切换到其它函数。当切换发生时，将正在执行的函数 1 的 R0~R15、CPSR 这 17 个寄存器临时保存起来，然后将希望执行的函数 2 的上次保存的数值恢复到 R0~R15、CPSR 这 17 个寄存器，这样芯片就从函数 1 切换到函数 2 运行了。当希望从函数 2 切换到函数 1 时，再将函数 2 的 17 个寄存器保存起来，恢复函数 1 的 17 个寄存器，芯片就又继续运行函数 1 了。这样便在函数 1 运行的中间插入了函数 2，这就是任务切换，也就是所谓的“上下文切换”，函数 1 或函数 2 所在的最上层父函数调用的一系列函数就组成了任务，任务是从最上层父函数开始运行的。

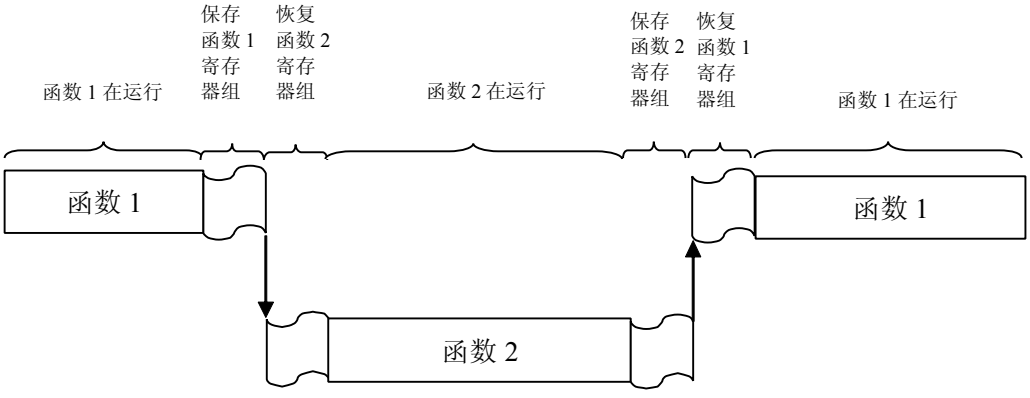


图 12　任务切换过程

这种切换也可以在多个任务之间进行，至于什么时候切换，怎么控制切换，这就是操作系统要做的事情了。

下面我们将遵循着这一设计思路来编写一个最简单的切换过程——2 个函数之间不停的互相切换，来验证任务切换过程中寄存器备份、恢复原理的正确性。

为了能看出任务切换的效果，我们设计 2 个函数 TEST_TestTask1 和 TEST_TestTask2，这两个函数都是死循环，反复执行“打印消息—>延迟”的过程，我们可以通过打印信息来确认是哪个函数在执行，伪码如下：

```
TEST_TestTask1：
while(1)
{
    打印“Task1 is running!”；
    延迟时间 1 秒;
}
```

```
TEST_TestTask2：
while(1)
{
    打印“Task2 is running!”；
    延迟时间 2 秒;
}
```

如果没有函数切换功能，那么这样的函数只要一开始执行，它们就会一直死循环执行下去，不会给其它函数执行的机会，我们就只能看到只有一个函数在循环打印消息。如果能够按照上面是所讲述的切换原理发生函数切换，那么我们就应该能看到的是这 2 个函数是在循环交替打印。

现在我们需要一个函数，它具有备份、恢复这 17 个寄存器的功能，在 TEST_TestTask1 和 TEST_TestTask2 需要切换时就调用它，完成上下文切换。我们将这个函数命名为 WLX_TaskSwitch，我们将 WLX_TaskSwitch 函数加入到 TEST_TestTask1 和 TEST_TestTask2 里：

```
TEST_TestTask1：
while(1)
{
    打印“Task1 is running!”；
    延迟时间 1 秒;
    WLX_TaskSwitch();
}
```

```
TEST_TestTask2：
while(1)
{
    打印“Task2 is running!”；
    延迟时间 2 秒;
    WLX_TaskSwitch();
}
```

我们将 WLX_TaskSwitch 函数设计为一个 C 函数，它仅对一些全局变量赋值，这些全局变量用来指明切换前函数的相关信息和切换后函数的相关信息。至于寄存器组备份、恢复的具体过程，由于是涉及到操作寄存器，因此只能使用汇编语言编写，将这个过程封装到由汇编语言编写的 WLX_ContextSwitch 函数里面来实现。

我们将使用 C 语言和汇编语言编写操作系统。C 语言作为高级语言具有较好的可移植性，并且控制硬件方便，在嵌入式领域有极广泛的应用，但无法直接控制芯片的寄存器。汇编语言是与芯片内部硬件息息相关的，可控制寄存器，但编码困难，可移植性差。因此，本手册本着尽可能使用 C 语言的原则，在 C 语言无法实现或实现成本太大的情况下才使用汇编语言。

现在我们先来看看 WLX_TaskSwitch 函数，最左侧的 5 位数字是代码在源代码文件里的行号。Wanlix 和 Mindows 的全部代码都可以从 http://blog.sina.com.cn/ifreecoding 网站免费下载，也可在网站内部的论坛上讨论。

```
00060  void WLX_TaskSwitch(void)
00061  {
00062      if(1 == guiCurTask)
00063      {
00064          /* 存入当前任务堆栈指针的地址 */
```

```
00065           gpuiCurTaskSpAddr = &guiTask1CurSp;
00066
00067           /* 获取即将运行任务的堆栈指针 */
00068           guiNextTaskSp = guiTask2CurSp;
00069
00070           /* 更新下次调度的任务 */
00071           guiCurTask = 2;
00072       }
00073       else //if(2 == guiCurTask)
00074       {
00075           gpuiCurTaskSpAddr = &guiTask2CurSp;
00076
00077           guiNextTaskSp = guiTask1CurSp;
00078
00079           guiCurTask = 1;
00080       }
00081
00082       /* 切换任务 */
00083       WLX_ContextSwitch();
00084   }
```

在这个函数里，我们用到了 guiCurTask、guiTask1CurSp、guiTask2CurSp、gpuiCurTaskSpAddr、guiNextTaskSp 这 5 个全局变量。guiCurTask 用来指示当前运行的任务，在 1 和 2 之间不断变化。guiTask1CurSp 保存的是 TEST_TestTask1 函数的寄存器组存储的内存地址，uiTask2CurSp 保存的是 TEST_TestTask2 函数的寄存器组存储的内存地址，寄存器组备份、恢复时用的就是这两个全局变量指向的内存空间。gpuiCurTaskSpAddr 用来存放 guiTask1CurSp 或 guiTask2CurSp 的地址，需要备份寄存器组的任务将指向它的寄存器组内存空间的变量的地址放入 gpuiCurTaskSpAddr 全局变量。guiNextTaskSp 存放的是 guiTask1CurSp 或 guiTask2CurSp，需要恢复寄存器组的任务将它的寄存器组内存空间地址放入 guiNextTaskSp 全局变量。这样，MDS_ContextSwitch 函数在寄存器组备份、恢复时就可以通过 gpuiCurTaskSpAddr 和 guiNextTaskSp 分别找到备份和恢复寄存器组的内存空间了。

下面详细解释一下 WLX_TaskSwitch 函数：

00062 行，对任务进行判断，如果当前运行的是任务 1 则进入此分支。

00065 行，运行到此行，说明当前运行的是任务 1，需要备份任务 1 的寄存器组数据，将任务 1 的全局变量 guiTask1CurSp 的地址存入全局变量 gpuiCurTaskSpAddr 中，准备供 MDS_ContextSwitch 函数使用。

00068 行，需要还原任务 2 的寄存器组数据，将任务 2 的全局变量 guiTask2CurSp 保存到全局变量 guiNextTaskSp 中，准备供 WLX_ContextSwitch 函数使用。

00071 行，准备从任务 1 切换到任务 2，将保存当前运行任务 ID 的全局变量 guiCurTask 更新为将要运行的任务 2。

00073~00080 行与 00062~00072 行功能类似，不通的是从任务 2 切换到任务 1 的过程。

00083 行，任务切换前的准备工作已经完成，调用 WLX_ContextSwitch 函数，开始寄存器组备份、恢复。

在介绍 WLX_ContextSwitch 函数之前，我们先设计一下保存寄存器组的内存结构，如图 13 所示：

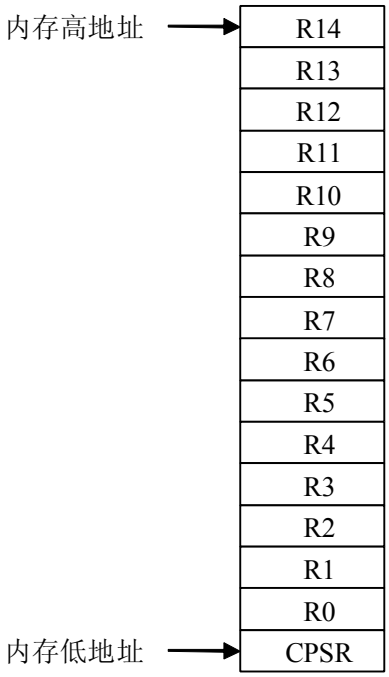


图 13　寄存器组在内存中的结构

寄存器组中每个写着寄存器名字的位置用来保存对应的寄存器，可以保存 R0~R14 和 CPSR 寄存器，但没有为 R15 进行备份，这是因为 Wanlix 的切换过程是由函数主动调用 WLX_ContextSwitch 函数实现的，是写死在代码里的，在编译的时候编译器就会安排代码，在任务切换前将 R15 自动保存在 R14 中，这样我们只需要备份 R14 就足够了。

这个寄存器组的位置存放在函数的栈中，当备份时，就从当前函数的 SP 栈指针指向的地址向栈增长的方向依次将寄存器存入，并更新 SP 栈指针，使之指向寄存器组中的“CPSR”，这也是一个压栈的过程，只不过是借用了函数的栈空间。当恢复时，从 SP 栈指针指向的寄存器组中依次恢复寄存器，并将 SP 栈指针恢复到函数切换前的所在位置，完成任务上下文切换。这时已经恢复的寄存器组所在的内存数据变为无效数据，当前函数运行时可能会压栈覆盖掉此空间的数据。

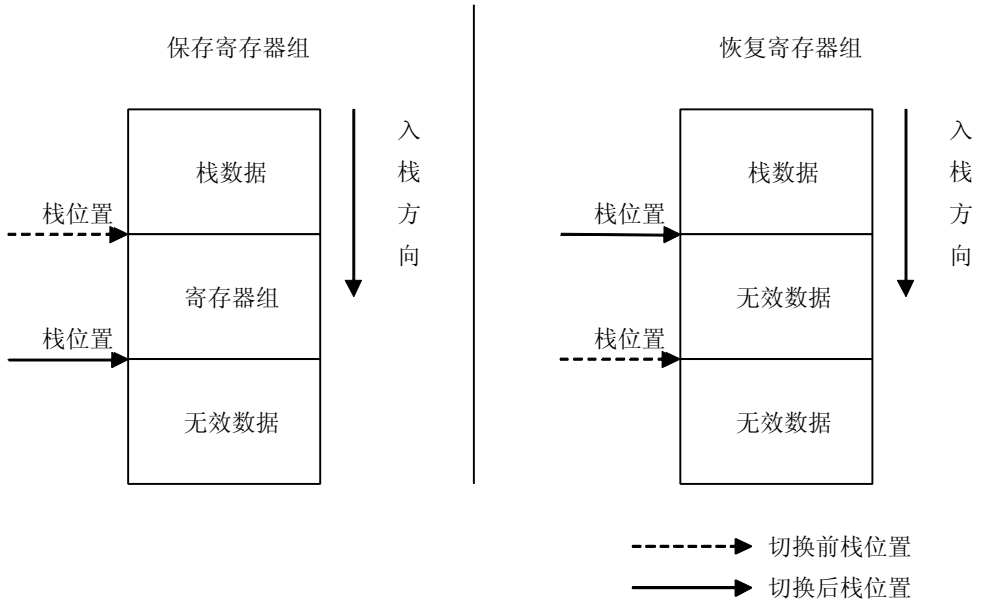


图 14　寄存器组在栈中的位置

在这里还需要说明一下进入操作系统前所使用的栈和进入操作系统后所使用的任务栈的情况。软件在刚启动时都是运行在没有操作系统的环境下，这时候所有函数都会使用同一个栈（ARM7 中中断模式除外），也就是从 main 函数开始，它及其它直接或间接调用的所有函数都是使用这个栈。进入操作系统状态后，程序就会以任务为功能单元运行了，在建立任务时需要为每个任务分配一个栈供任务运行时使用，这个栈就称之为任务栈。软件进入操作系统后就不使用原有共用的栈了，每个任务完全使用自己私有的任务栈，如下图所示：

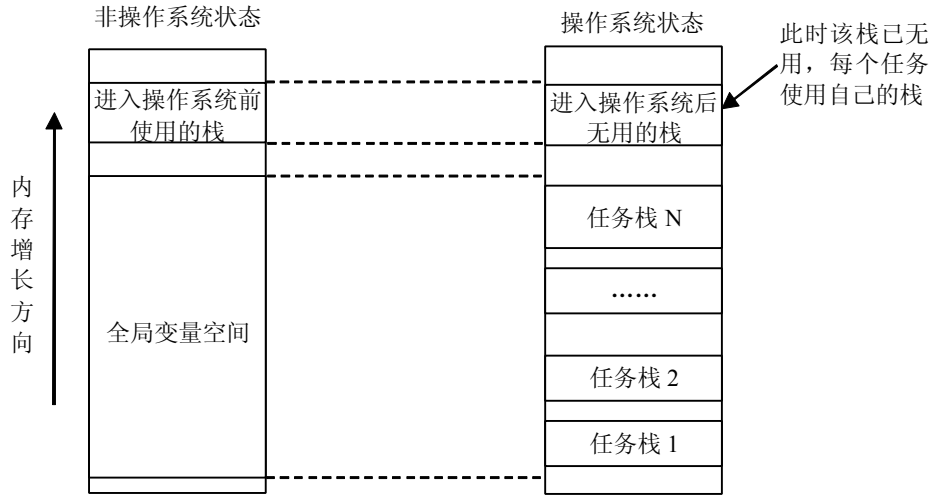


图 15　进入操作系统前后的栈空间使用情况

目前任务栈都是使用全局变量静态申请的，因此在上图中可以看到每个任务栈都对应着全局变量空间。另外，由于操作系统不会重新返回到非操作系统状态下，因此在进入操作系统前的栈中保存的数据也没有用处了，用户可以将这个栈的内存空间拿来另作它用。

有了前面的铺垫，我们来看看操作系统内核最核心的函数 WLX_ContextSwitch：

```
00012        .func WLX_ContextSwitch
```

```
00013  WLX_ContextSwitch:
00014
00015      @保存当前任务的堆栈信息
00016      STMDB  R13, {R0-R14}
00017      SUB    R13, R13, #0x3C
00018      MRS    R0, CPSR
00019      STMDB  R13!, {R0}
00020
00021      @保存当前任务的指针值
00022      LDR    R0, =gpuiCurTaskSpAddr
00023      LDR    R1, [R0]
00024      CMP    R1, #0
00025      BEQ    GETNEXTTASKSP
00026      STR    R13, [R1]
00027
00028  GETNEXTTASKSP:
00029      @获取将要运行任务的堆栈信息并运行新任务
00030      LDR    R0, =guiNextTaskSp
00031      LDR    R13, [R0]
00032      LDMIA  R13!, {R0}
00033      MSR    CPSR, R0
00034      LDMIA  R13, {R0-R14}
00035      BX     R14
00036
00037      .endfunc
```

00012 行，定义 WLX_ContextSwitch 函数。

00013 行，这是一条汇编伪指令，只是一个标号，代表 WLX_ContextSwitch 函数的起始地址，并不生成可执行代码。

00015 行，在 GNU 环境的汇编语言里，“@”符号代表注释符，编译时其后的所有字符都被注释掉，不生成任何代码，其存在只为程序提供说明性帮助。

00016 行，这是该函数的第一条可执行语句，它将当前任务的 R0-R14 寄存器保存到寄存器组中，寄存器组的地址由 SP 寄存器指定。

00017 行，更新数据入栈后的 SP 栈指针。

00018 行，将当前任务的 CPSR 寄存器保存到 R0 中。

00019 行，将 R0 寄存器存入寄存器组，也就是把 CPSR 寄存器的内容存入寄存器组，并更新 SP 寄存器。00016~00019 行所使用的汇编指令并不会改变 CPSR 寄存器的内容，因此 00019 行这条指令保存的 R0 就是当前任务进入 WLX_ContextSwitch 函数前的 CPSR 寄存器的数值。

00022 行，获取全局变量 gpuiCurTaskSpAddr 的地址。

00023 行，获取全局变量 gpuiCurTaskSpAddr 的内容，也就是存放当前任务的寄存器组的全局变量 guiTaskXCurSp（当前任务为 1 则 guiTaskXCurSp 为 guiTask1CurSp，当前任务为 2 则则 guiTaskXCurSp 为 guiTask2CurSp）。

00024 行，将 gpuiCurTaskSpAddr 全局变量的内容与 0 做比较。在非操作系统状态下全局变量 gpuiCurTaskSpAddr 为 0，无需保存寄存器组数据。

00025 行，如果 gpuiCurTaskSpAddr 全局变量为 0，则跳转到 GETNEXTTASKSP 标志所在地址，准备进入操作系统状态，但还没有任务在运行，因此，不需要保存非操作系统状态下的 SP 栈指针。如果不为 0 则代表已经进入操作系统状态，不执行本行，执行 00026 行。

00026 行，将当前任务的 SP 栈指针存入当前任务的 guiTaskXCurSp 全局变量中，下次运行时可借此找到任务的栈指针，并根据栈指针从寄存器组中恢复出寄存器的数值。

00030 行，获取全局变量 guiNextTaskSp 的地址。

00031 行，获取全局变量 guiNextTaskSp 的内容，也就是存放将要运行任务的寄存器组的地址，将寄存器组的地址存入 SP 中。

00032 行，根据 SP 从寄存器组中恢复将要运行任务的 CPSR 寄存器，恢复到 R0 中。

00033 行，将 R0 保存到 CPSR 中，也就是恢复了将要运行任务的 CPSR 寄存器。

00034 行，恢复 R0-R14 寄存器，其中包含了 SP 寄存器，因此不再需要额外更新 SP 寄存器。

00035 行，跳转到将要运行的任务。至此，已经完成了 2 个任务的寄存器出入栈工作，芯片当前工作的寄存器已经从切换前运行的任务全部换成了切换后将要运行的任务，寄存器数据已经全部处理完了，只要能将 PC 指针正确跳到将要运行任务上次切出去那一时刻的位置就可以了。此时 LR 寄存器中保存的就是将要运行任务的上次切出去的地址，跳转到 LR，完成 2 个任务切换的最后一步。

上面实现了任务的切换过程，但是还有一些事情需要解决，那就是测试函数 TEST_TestTask1 和 TEST_TestTask2 第一次运行时，栈中的寄存器组是空的，无法进行寄存器组恢复，也就无法切换了。因此，我们需要一个初始化函数，来为第一次运行的任务初始化它的寄存器组栈空间。这个任务初始化函数是 WLX_TaskInit，它只需要在本任务的栈内为寄存器组赋初值就可以了，至于每个寄存器应该赋什么初值，我们可以分析一下。

由于任务是第一次运行，所有一切都是空的，函数不会从 R0~R12 寄存器读数据使用，因此可以将这些寄存器全部置 0。SP 是栈指针，所以需要将任务的栈顶赋给 SP。LR 是返回地址，需要通过跳转到 LR 去第一次执行这个函数，而函数名就是函数的第一条指令所在的地址，函数名是指针，因此，将函数名存入存入 LR 中。

具体实现过程来看下面来看代码：

```
00020  void WLX_TaskInit(U8 ucTask, VFUNC vfFuncPointer, U32* puiTaskStack)
00021  {
00022      U32* puiSp;
00023
00024      /* 对堆栈初始化 */
00025      puiSp = puiTaskStack;               /* 获取堆栈指针 */
00026
00027      *(--puiSp) = (U32)vfFuncPointer;    /* R14 */
00028      *(--puiSp) = (U32)puiTaskStack;     /* R13 */
00029      *(--puiSp) = 0;                     /* R12 */
00030      *(--puiSp) = 0;                     /* R11 */
00031      *(--puiSp) = 0;                     /* R10 */
00032      *(--puiSp) = 0;                     /* R9 */
00033      *(--puiSp) = 0;                     /* R8 */
00034      *(--puiSp) = 0;                     /* R7 */
00035      *(--puiSp) = 0;                     /* R6 */
00036      *(--puiSp) = 0;                     /* R5 */
00037      *(--puiSp) = 0;                     /* R4 */
00038      *(--puiSp) = 0;                     /* R3 */
00039      *(--puiSp) = 0;                     /* R2 */
00040      *(--puiSp) = 0;                     /* R1 */
00041      *(--puiSp) = 0;                     /* R0 */
00042      *(--puiSp) = MODE_USR;              /* CPSR */
00043
00044      /* 记录当前任务的堆栈指针，下次运行这个任务时可根据该值恢复堆栈 */
00045      if(1 == ucTask)
00046      {
00047          guiTask1CurSp = (U32)puiSp;
00048      }
```

```
00049       else //if(2 == ucTask)
00050       {
00051           guiTask2CurSp = (U32)puiSp;
00052       }
00053   }
```

00020 行，函数定义，ucTask 入口参数确定需要初始化的函数；vfFuncPointer 入口参数是需要初始化任务的函数；puiTaskStack 入口参数是这个任务所使用的任务栈指针，需要是栈顶满栈指针。

00025 行，将 puiSp 变量指向任务栈栈顶。

00027 行，将函数指针存入寄存器组的 LR 位置。当该任务第一次运行，恢复寄存器时，该函数指针就会被恢复到 LR 寄存器中，当程序跳转到 LR 时也就开始运行该函数了。

00028 行，将任务栈栈顶地址存入寄存器组的 SP 位置。当该任务第一次运行时，栈内初始化的数据全部取出后，任务栈已经空了，此时 SP 应该指向栈顶，这时候函数才开始运行。将任务栈栈顶地址存入 SP 中，当该任务第一次运行，恢复寄存器时，任务栈栈顶地址就会被恢复到 SP 寄存器中，该任务就会从 SP 所指的地址开始存放栈数据，也就到达了控制任务栈的目的。

00029~00041 行，任务刚创建时 R0~R12 寄存器中数据为无效值，因此此处全部填 0。

00042 行，将 USR 模式存入寄存器组的 CPSR 位置。这样当该任务第一次运行时，USR 模式就会被恢复到 CPSR 寄存器中，任务就会从 USR 模式开始启动。MODE_USR 是一个宏定义，其值为 0x10，可以参考图 5，代表 USR 模式。函数第一次运行时 CPSR 里面 NZCV 等各种状态均为 0，因此此处 CPSR 寄存器中的状态位均被初始化为 0。

00045~00052 行，将每个任务的栈信息保存到它对应的全局变量中，供任务切换时使用。

创建任务所使用的函数都是通过 WLX_TaskInit 函数以这种隐式的方式开始运行的，并没有直接调用。在创建任务时就确定了任务所使用的根函数以及栈等其它信息，后续随着我们对操作系统的不断完善，还会有更多的信息被加入到任务中来，这也是任务不同于函数的地方，任务比函数拥有更多的信息，这样，操作系统才可以利用这些信息更方便的管理任务调度。

最后需要使用 WLX_TaskStart 函数从非操作系统状态开始进入到操作系统状态。WLX_TaskStart 函数很简单，就是对上面介绍过的全局变量做一些初始化，然后调用任务切换函数 WLX_TaskSwitch，WLX_TaskSwitch 函数再调用 WLX_ContextSwitch 函数，将任务初始化函数 WLX_TaskInit 初始化的参数恢复到寄存器中开始运行，开始了第一个任务的运行，这样就进入到了操作系统状态。

WLX_TaskStart 函数很简单，不再详细解释，代码如下：

```
00091   void WLX_TaskStart(void)
00092   {
00093       /* 任务运行前不需要保存任务堆栈指针 */
00094       gpuiCurTaskSpAddr = (U32*)NULL;
00095
00096       /* 获取即将运行任务的堆栈指针 */
00097       guiNextTaskSp = guiTask1CurSp;
00098
00099       /* 更新下次调度的任务 */
00100       guiCurTask = 1;
00101
00102       WLX_ContextSwitch();
00103   }
```

现在我们已经完成任务切换所需要的全部代码，在 main 函数里首先初始化硬件，然后调用 WLX_TaskInit 函数对 2 个任务进行初始化，最后调用 WLX_TaskStart 函数启动任务调度，这 2 个任务就开始交替执行了，交替向串口打印数据。

```
00014  S32 main(void)
00015  {
00016      /* 初始化硬件 */
00017      DEV_HardwareInit();
00018
00019      /* 创建任务 */
00020      WLX_TaskInit(1, TEST_TestTask1, TEST_GetTaskInitSp(1));
00021      WLX_TaskInit(2, TEST_TestTask2, TEST_GetTaskInitSp(2));
00022
00023      /* 开始任务调度 */
00024      WLX_TaskStart();
00025
00026      return 0;
00027  }
```

测试函数 TEST_TestTask1 和 TEST_TestTask2 运行时函数调用关系如下：

—>TEST_TestTask1
  —>WLX_TaskSwitch
    —>WLX_ContextSwitch
      —> TEST_TestTask2
        —>WLX_TaskSwitch
          —>WLX_ContextSwitch
            —>TEST_TestTask1
              —>……

我们来看看最终的效果：

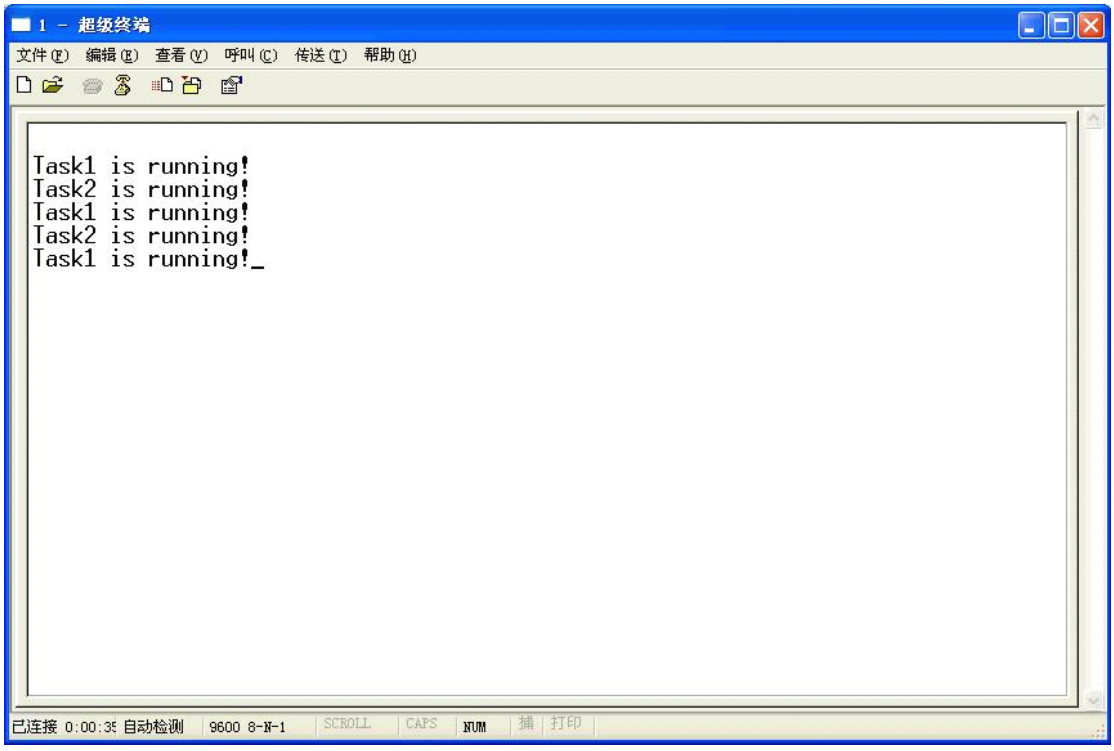


图 16　两个任务交替执行

通过图 16 我们可以看到这两个任务交替的运行，在代码里我们并没有直接运行 TEST_TestTask1 和 TEST_TestTask2 函数，而是采用操作系统创建任务、切换任务的原理运行这两个函数的。从串口工具的输出来看，已经完全实现了我们的设计！

读者还可以观看串口输出的视频，请登陆 http://blog.sina.com.cn/ifreecoding 网站下载，该视频动态的记录了两个任务在串口工具上输出的过程，可以看到 TEST_TestTask1 任务执行 1 秒后切换到 TEST_TestTask2 任务，TEST_TestTask2 任务执行 2 秒后切换到 TEST_TestTask1 任务，如此循环。

还有一点需要说明，某些函数具有返回值，但我并没有完全判断这些返回值，当不影响软件功能时我一般是采用 viod 屏蔽了函数返回值，以突出本手册介绍的重点。但我建议，如果你是在做一个项目的话，最好能判断函数的返回值，以增强系统的健壮性。

第 2 节 任意任务间的切换

上一节我们使用 2 个固定的任务验证了操作系统任务切换的功能，但这些代码并不具有通用性，如果要扩充其它任务，就必须修改操作系统函数，这显然是不可接受的。操作系统作为独立于用户代码的部分，它的内部细节应该是不被用户所见的，是一个黑盒，需要做到用户只需要修改操作系统提供的接口文件里面的参数，调用接口函数就可以完全满足程序开发的要求。因此，在本节我们将对上节的代码做些改动，使其可以支持任意多个任务之间的互相切换，而又不需要修改 Wanlix 目录下的操作系统代码，仅仅是编写 srccode 目录下的用户代码，调用操作系统的接口函数即可，这样才真正实现了操作系统的独立性。

首先我们来看看任务切换函数——WLX_TaskSwitch。上节中，这个函数固定在两个任务之间切换，因此要实现可以切换到任何一个函数的功能就必须修改此函数，需要为这个函数增加一个入口参数，用这个入口参数来指明需要切换到的任务。WLX_TaskSwitch 函数的主要功能是做好任务切换前的准备，将当前运行任务的栈指针和将要运行任务的栈指针存入到对应的全局变量中，上节中，为每个任务分别指定了 guiTask21CurSp 和 guiTask2CurSp 全局变量保存它们的当前栈指针，每个全局变量绑定到了任务，因此，这个入口参数还必须能够关联到任务的栈指针。

为此，我们引入 TCB 的概念，在操作系统里这是一个非常重要的概念。TCB 是 Task Control Block 的缩写，意为任务控制块，与任务控制相关的重要信息都放被到 TCB 里。TCB 是一个结构体，每个任务都拥有一个 TCB ，可以把每个任务的与任务控制相关的结构都放入到它的 TCB 中，因此我们可以将任务当前的栈指针保存到它的 TCB 中，到目前为止，TCB 格式如下：

```
typedef struct w_tcb
{
    U32 uiTaskCurSp;
}W_TCB;
```

TCB 结构不只是这么简单，只是到目前为止就是这么简单，随着操作系统功能的不断完善，TCB 也会不断的增加它的结构。

我们可以考虑将 TCB 放到任务的栈中。当任务创建时在栈的开始处保留一块内存作为

TCB 的存放空间，TCB 之后的栈空间才作为真正的栈使用，这样，任务的 TCB 也就与任务绑定到了一起，每个 TCB 就可以代表一个任务。由于 ARM 芯片是线性地址空间，也就是说每个内存地址都是唯一的，因此每个任务堆栈的开始地址也就是唯一的，因此每个 TCB 的地址也就是唯一的了，这样我们就可以使用 TCB 的地址来代表各个不同的任务。

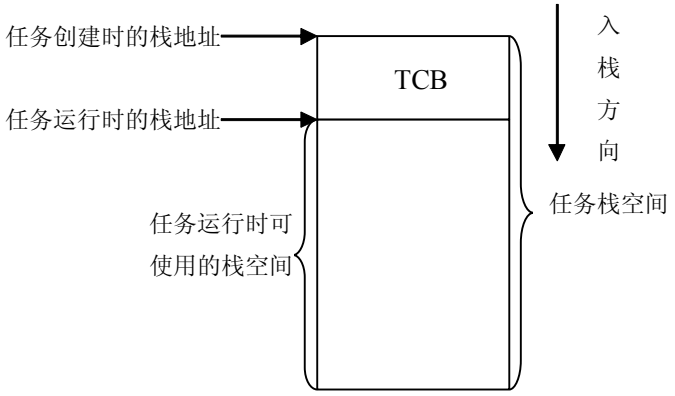


图 17　TCB 在栈中的位置

有了 TCB，下面我们来修改 WLX_TaskSwitch 函数。修改很简单，只是将 TCB 指针作为入口参数，在函数里替换掉原来与任务相关的全局变量，修改后的函数如下：

```
00107  void WLX_TaskSwitch(W_TCB* pstrTcb)
00108  {
00109      /* 保存当前任务堆栈指针的地址，在汇编语言可以通过这个变量写入任务切换前最后时刻
00110         的堆栈地址 */
00111      gpuiCurTaskSpAddr = &gpstrCurTcb->uiTaskCurSp;
00112
00113      /* 保存即将运行任务的堆栈指针 */
00114      guiNextTaskSp = pstrTcb->uiTaskCurSp;
00115
00116      /* 保存即将运行任务的 TCB */
00117      gpstrCurTcb = pstrTcb;
00118
00119      WLX_ContextSwitch();
00120  }
```

00107 行，入口参数 pstrTcb 是将要运行任务的 TCB 指针，准备切换到该任务运行。

00111 行，将当前运行任务的 TCB 中保存当前栈指针变量的地址存入全局变量 gpuiCurTaskSpAddr 中。

00114 行，将将要运行任务的 TCB 中保存当前栈指针的变量，也就是当前的栈指针，存入全局变量 guiNextTaskSp 中。

00117 行，将全局变量 gpstrCurTcb 更新为将要运行任务的 TCB，为下次任务切换做准备。

00119 行，调用汇编函数 WLX_ContextSwitch 执行具体的寄存器备份、恢复操作。

同理，WLX_TaskStart 函数也需要做类似的变量替换，不再介绍，读者自行分析。

```
00127  void WLX_TaskStart(W_TCB* pstrTcb)
00128  {
00129      /* 保存即将运行任务的堆栈指针 */
00130      guiNextTaskSp = pstrTcb->uiTaskCurSp;
00131
```

```
00132       /* 保存即将运行任务的 TCB */
00133       gpstrCurTcb = pstrTcb;
00134
00135       WLX_SwitchToTask();
00136   }
```

由于增加了 TCB，因此必须修改任务初始化函数。从本节开始，所有的任务将采用 WLX_TaskCreate 函数创建，在 WLX_TaskCreate 函数内分别对 TCB 和栈进行初始化。

```
00018   W_TCB* WLX_TaskCreate(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize)
00019   {
00020       W_TCB* pstrTcb;
00021
00022       /* 对创建任务所使用函数的指针合法性进行检查 */
00023       if(NULL == vfFuncPointer)
00024       {
00025           /* 指针为空，返回失败 */
00026           return (W_TCB*)NULL;
00027       }
00028
00029       /* 对任务堆栈合法性进行检查 */
00030       if((NULL == pucTaskStack) || (0 == uiStackSize))
00031       {
00032           /* 堆栈不合法，返回失败 */
00033           return (W_TCB*)NULL;
00034       }
00035
00036       /* 初始化 TCB */
00037       pstrTcb = WLX_TaskTcbInit(pucTaskStack, uiStackSize);
00038
00039       /* 初始化任务堆栈 */
00040       WLX_TaskStackInit(pstrTcb, vfFuncPointer);
00041
00042       return pstrTcb;
00043   }
```

00018 行，函数返回值是被创建任务的 TCB 指针；入口参数 vfFuncPointer 是创建任务所使用的函数；入口参数 pucTaskStack 是创建任务所使用的栈地址，是栈的低地址；入口参数 uiStackSize 是栈的大小。

00023 行，对入口参数判断，如果函数指针为空，则返回 NULL 空指针代表创建任务失败。在 C 语言里，指针为 NULL（也就是 0）代表无效指针，因为 0 地址的内存一般都是中断向量表的复位向量，正常使用指针时是不会指向这里的。

00030 行，对入口参数判断，如果栈指针为 NULL 或者栈大小为 0，则返回失败。

00037 行，调用 WLX_TaskTcbInit 函数初始化任务的 TCB，并得到当前任务的 TCB 指针。

00040 行，调用 WLX_TaskStackInit 函数初始化当前的任务栈。

00042 行，任务创建成功，返回当前任务的 TCB。以后就可以使用这个返回值来代表这个任务了。

目前的 TCB 比较简单，只有一个保存栈地址的变量，在 WLX_TaskTcbInit 函数里就是来初始化这个变量的，并返回 TCB 指针。

```
00051   W_TCB* WLX_TaskTcbInit(U8* pucTaskStack, U32 uiStackSize)
```

```
00052  {
00053      W_TCB* pstrTcb;
00054      U8* pucStackBy4;
00055
00056      /* 堆栈满地址，需要 4 字节对齐 */
00057      pucStackBy4 = (U8*)(((U32)pucTaskStack + uiStackSize) & 0xFFFFFFFC);
00058
00059      /* TCB 结构存放的地址，需要 4 字节对齐 */
00060      pstrTcb = (W_TCB*)(((U32)pucStackBy4 - sizeof(W_TCB)) & 0xFFFFFFFC);
00061
00062      /* 初始化 TCB 结构 */
00063      pstrTcb->uiTaskCurSp = (U32)pstrTcb;
00064
00065      return pstrTcb;
00066  }
```

00051 行，函数返回值是被创建任务的 TCB 指针，创建任务后这个 TCB 就代表该任务；pucTaskStack 是任务的栈地址，是栈的低地址；uiStackSize 是栈的大小。

00057 行，确定栈顶地址。“(U32)pucTaskStack + uiStackSize”是栈顶地址，由于栈必须是 4 字节对齐，因此再通过“& 0xFFFFFFFC”操作，从栈顶向下寻找 4 字节对齐的地址作为栈顶地址。

00060 行，确定 TCB 地址。“(U32)ucStackBy4 - sizeof(W_TCB)”操作，从栈中减去存放 TCB 的空间，再通过“& 0xFFFFFFFC”操作，向下寻找 4 字节对齐的地址作为存放 TCB 的起始地址，这个也是任务调度时使用的栈顶地址。

00063 行，初始化 TCB 中的变量，保存任务的栈指针。

00065 行，返回任务的 TCB 指针。

TCB 的引入也需要对 WLX_TaskStackInit 函数做简单的修改，由于该函数只是简单使用 TCB 替换了原来专用的全局变量，没有大的改动，就不做过多介绍了，读者可以对比上节的函数自己分析。

```
00074  void WLX_TaskStackInit(W_TCB* pstrTcb, VFUNC vfFuncPointer)
00075  {
00076      U32* puiSp;
00077
00078      /* 对堆栈初始化 */
00079      puiSp = (U32*)pstrTcb->uiTaskCurSp; /* 获取存放变量的堆栈指针 */
00080
00081      *(--puiSp) = (U32)vfFuncPointer;    /* R14 */
00082      *(--puiSp) = pstrTcb->uiTaskCurSp;  /* R13 */
00083      *(--puiSp) = 0;                     /* R12 */
00084      *(--puiSp) = 0;                     /* R11 */
00085      *(--puiSp) = 0;                     /* R10 */
00086      *(--puiSp) = 0;                     /* R9 */
00087      *(--puiSp) = 0;                     /* R8 */
00088      *(--puiSp) = 0;                     /* R7 */
00089      *(--puiSp) = 0;                     /* R6 */
00090      *(--puiSp) = 0;                     /* R5 */
00091      *(--puiSp) = 0;                     /* R4 */
00092      *(--puiSp) = 0;                     /* R3 */
00093      *(--puiSp) = 0;                     /* R2 */
00094      *(--puiSp) = 0;                     /* R1 */
00095      *(--puiSp) = 0;                     /* R0 */
00096      *(--puiSp) = MODE_USR;              /* CPSR */
00097
```

```
00098       /* 记录当前任务的堆栈指针，下次运行这个任务时可根据该值恢复堆栈 */
00099       pstrTcb->uiTaskCurSp = (U32)puiSp;
00100   }
```

经过上述修改操作系统就具有通用性了，无论建立多少个任务都无需修改操作系统的代码，只要为任务分配一个栈空间，使用 WLX_TaskCreate 函数就可以创建任务，并可以使用这个任务的 TCB 指针作为入口参数，调用 WLX_TaskSwitch 函数就可以切换到这个任务。

另外说一点，创建任务时，需要用户先用全局变量为所创建的任务申请一个任务栈空间，将它的起始地址和大小作为参数传递给 WLX_TaskCreate 函数来创建任务。如果能将申请任务栈的操作封装到 WLX_TaskCreate 函数里面就会更方便一些，但我在 GNU 环境下没有找到配置堆（heap）的方法（谁知道请在论坛上反馈一下，谢谢！），因此无法在 WLX_TaskCreate 函数里使用 C 函数库里的 malloc 函数从堆中申请任务栈。如果使用自己编写的堆函数则不如 C 库函数的方便，兼容性也不好，因此这里需要用户自己申请任务的栈空间。后面在 Cortex 内核芯片上，我们将换一个编译器，到那时候再完善这个功能。

在看最终效果前，我们再对任务切换过程中寄存器备份、恢复操作做最后一点优化。上节我们使用 WLX_ContextSwitch 函数完成任务寄存器入栈、出栈及最后跳转的操作，这样做存在 2 个问题：

1. 每次执行任务切换时都需要多执行 2 条汇编指令，来判断是否是从非操作系统状态切换到操作系统状态，请参考上节 WLX_ContextSwitch 函数的 00024 行和 00025 行。
2. 在程序从非操作系统状态切换为操作系统状态时，没有必要将芯片寄存器保存到系统栈中。

为此，我们将原有的 WLX_ContextSwitch 函数拆分成 2 个函数，WLX_ContextSwitch 函数和 WLX_SwitchToTask 函数。WLX_ContextSwitch 函数仍被 WLX_TaskSwitch 函数调用，用于每次任务切换，WLX_SwitchToTask 函数被 WLX_TaskStart 函数调用，用于第一次任务切换。

这两个汇编函数没有实质性的改动，不再详细介绍，请读者自行分析。

```
00012       .func WLX_ContextSwitch
00013   WLX_ContextSwitch:
00014
00015       @保存当前任务的堆栈信息
00016       STMDB   R13, {R0-R14}
00017       SUB     R13, R13, #0x3C
00018       MRS     R0, CPSR
00019       STMDB   R13!, {R0}
00020
00021       @保存当前任务的指针值
00022       LDR     R0, =gpuiCurTaskSpAddr
00023       LDR     R1, [R0]
00024       STR     R13, [R1]
00025
00026       @获取将要运行任务的指针
00027       LDR     R0, =guiNextTaskSp
00028       LDR     R13, [R0]
00029
00030       @获取将要运行任务的堆栈信息并运行新任务
00031       LDMIA   R13!, {R0}
00032       MSR     CPSR, R0
```

```
00033      LDMIA  R13, {R0-R14}
00034      BX     R14
00035
00036      .endfunc


00043      .func WLX_SwitchToTask
00044  WLX_SwitchToTask:
00045
00046      @获取将要运行任务的指针
00047      LDR    R0, =guiNextTaskSp
00048      LDR    R13, [R0]
00049
00050      @获取将要运行任务的堆栈信息并运行新任务
00051      LDMIA  R13!, {R0}
00052      MSR    CPSR, R0
00053      LDMIA  R13, {R0-R14}
00054      BX     R14
00055
00056      .endfunc
```

至此就完成了本节代码的修改。在测试代码里我们建立 3 个任务，TEST_TestTask1、TEST_TestTask2 和 TEST_TestTask3，并将它们的 TCB 保存到全局变量 gpstrTask1Tcb、gpstrTask1Tcb 和 gpstrTask1Tcb 中，供任务切换时使用。

```
00020  S32 main(void)
00021  {
00022      /* 初始化硬件 */
00023      DEV_HardwareInit();
00024
00025      /* 创建任务 */
00026      gpstrTask1Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask1, gaucTask1Stack,
00027                                     TASKSTACK);
00028      gpstrTask2Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask2, gaucTask2Stack,
00029                                     TASKSTACK);
00030      gpstrTask3Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask3, gaucTask3Stack,
00031                                     TASKSTACK);
00032
00033      /* 开始任务调度, 从任务 1 开始执行 */
00034      WLX_TaskStart(gpstrTask1Tcb);
00035
00036      return 0;
00037  }

00044  void TEST_TestTask1(void)
00045  {
00046      while(1)
00047      {
00048          DEV_PutString((U8*)"\r\nTask1 is running!");
00049
00050          DEV_DelayMs(1000);              /* 延迟 1s */
00051
00052          WLX_TaskSwitch(gpstrTask3Tcb);  /* 任务切换 */
00053      }
00054  }

00061  void TEST_TestTask2(void)
```

```
00062  {
00063      while(1)
00064      {
00065          DEV_PutString((U8*)"\r\nTask2 is running!");
00066
00067          DEV_DelayMs(2000);              /* 延迟 2s */
00068
00069          WLX_TaskSwitch(gpstrTask1Tcb);  /* 任务切换 */
00070      }
00071  }
```

```
00078  void TEST_TestTask3(void)
00079  {
00080      while(1)
00081      {
00082          DEV_PutString((U8*)"\r\nTask3 is running!");
00083
00084          DEV_DelayMs(3000);              /* 延迟 3s */
00085
00086          WLX_TaskSwitch(gpstrTask2Tcb);  /* 任务切换 */
00087      }
00088  }
```

这 3 个任务在循环运行，向串口打印数据，每间隔一段时间切换到另外一个任务继续运行。TEST_TestTask1 任务运行时向串口打印“Task1 is running!”代表 TEST_TestTask1 任务开始运行，1 秒后主动切换到 TEST_TestTask3 任务，TEST_TestTask3 任务运行时向串口打印“Task3 is running!”代表 TEST_TestTask3 任务开始运行，3 秒后主动切换到 TEST_TestTask2 任务，TEST_TestTask2 任务运行时向串口打印“Task2 is running!”代表 TEST_TestTask2 任务开始运行，2 秒后主动切换到 TEST_TestTask1 任务，如此反复循环。

编译本节代码，串口打印如下图：

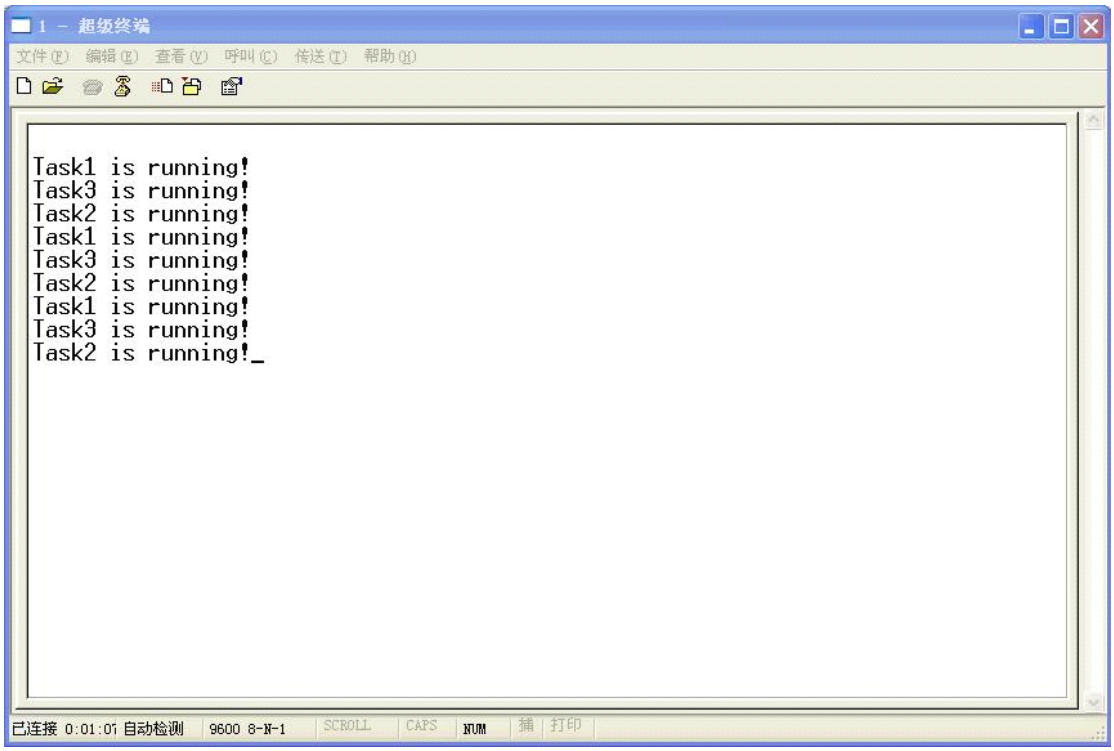


图 18　可创建任意多个任务的运行结果

读者也可通过本节的视频观察这 3 个任务的动态执行过程，读者也可以自行增加任务在单板上运行一下，体验本节的收获！

第 3 节 用户代码入口——根任务

经过上节的修改，Wanlix 操作系统可以建立任意多个任务，但是在操作系统运行之前必须得先建立一个任务，然后再调用 WLX_TaskStart 函数从非操作系统状态切换到操作系统状态，如果没有这么做的话系统就会崩溃。这一过程需要用户在用户代码里完成，相当于使用用户代码来初始化操作系统，这无疑给用户增加了一个限制，也不利于用户使用。

为了解决这个问题，我们提出操作系统“根”任务的概念，所谓“根”任务，它是其它所有任务的“根”，其它所有的任务都是从这个根任务开始的，我们将之命名为 WLX_RootTask。

我们在 main 函数里首先建立根任务，然后调用 WLX_TaskStart 函数切换到操作系统状态，去执行根任务，将根任务作为留给用户的接口，main 函数则被封装到操作系统内部，用户不可见，用户只要认为自己的代码是从根任务开始的就可以了，这样，在用户代码执行前，操作系统就已经可以使用了，这个问题也就解决了。

为此，我们需要将 main 函数从原来的 test.c 文件中搬移到 wlx_core_c.c 文件中，将它封装到操作系统内部，作为操作系统的一部分。

```
00019  S32 main(void)
00020  {
00021      /* 创建根任务 */
00022      gpstrRootTaskTcb = WLX_TaskCreate((VFUNC)WLX_RootTask, gaucRootTaskStack,
00023                                        ROOTTASKSTACK);
00024
00025      /* 开始任务调度，从根任务开始执行 */
00026      WLX_TaskStart(gpstrRootTaskTcb);
00027
00028      return 0;
00029  }
```

在 main 函数运行完毕后就开始运行根任务 WLX_RootTask 了，用户可以在 WLX_RootTask 任务中创建自己的任务，我们将上节的例子移植过来，只需要将原有 main 函数中创建任务的用户代码移植到根任务 WLX_RootTask 中就可以了。

```
00010  void WLX_RootTask(void)
00011  {
00012      /* 初始化硬件 */
00013      DEV_HardwareInit();
00014
00015      /* 创建任务 */
00016      gpstrTask1Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask1, gaucTask1Stack,
00017                                     TASKSTACK);
00018      gpstrTask2Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask2, gaucTask2Stack,
00019                                     TASKSTACK);
00020      gpstrTask3Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask3, gaucTask3Stack,
00021                                     TASKSTACK);
00022
00023      WLX_TaskSwitch(gpstrTask1Tcb);      /* 任务切换 */
00024  }
```

根任务 WLX_RootTask 虽然是操作系统建立的任务，属于操作系统的一部分，但它的内容却完全需要用户编写，因此将它放到 wlx_userboot.c 文件中，将 wlx_userboot.c 文件放到用户代码目录 srccode 中，与用户代码绑定在一起。

编译本节代码，运行，串口打印如下。读者也可通过本节的视频观察这 3 个任务动态执行过程，虽然实现上与上节不同，但输出结果却是一样的。

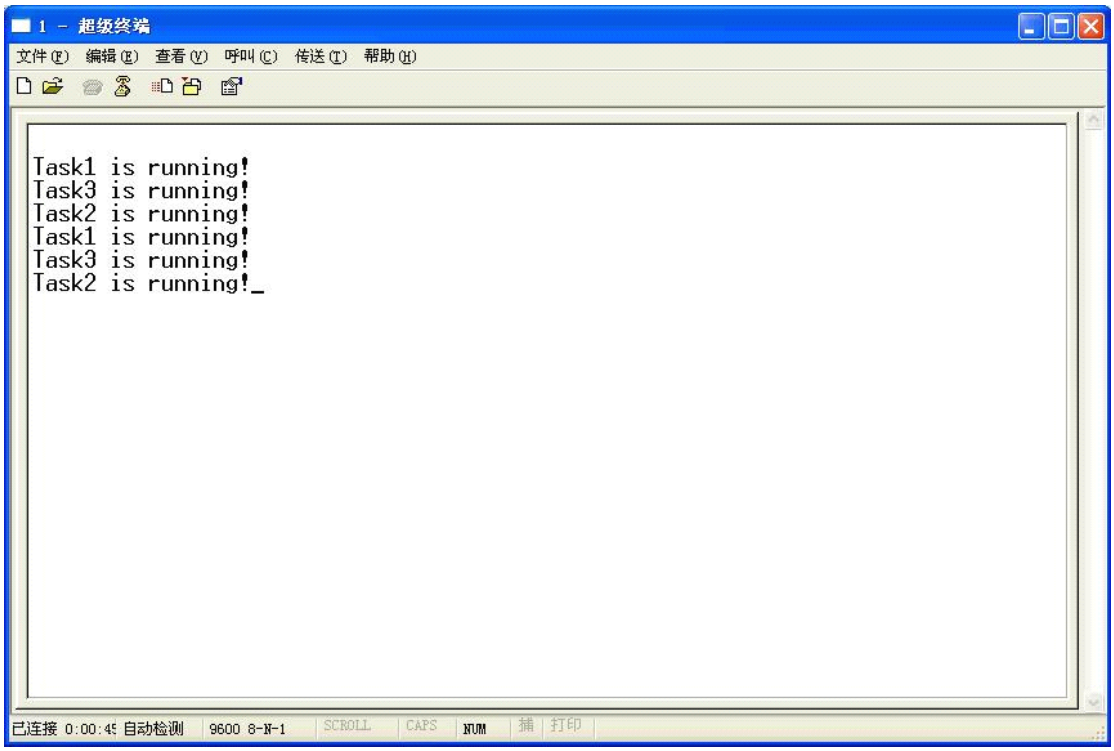


图 19　使用根任务作为用户入口的运行结果

第 4 节 使用 Wanlix 编写交通红绿灯控制系统

至此我们已经实现了一个非常简单、小巧的操作系统——Wanlix，简单到它只具备任务切换这一项任务管理功能，而且需要用户自己主动切换，实时性较差。但无论如何，它确实是实现了任务的切换，这是不争的事实，从前面打印的例子就可以证明。

本节我们将使用 Wanlix 开发一个交通红绿灯的控制程序，通过这个稍微复杂点的程序来应用 Wanlix 操作系统。

首先，先来了解一下这个交通红绿灯的功能，然后再设计软件结构、编码，最后在单板上运行，观察结果，展示使用 Wanlix 操作系统开发的第一个嵌入式系统。

功能说明：

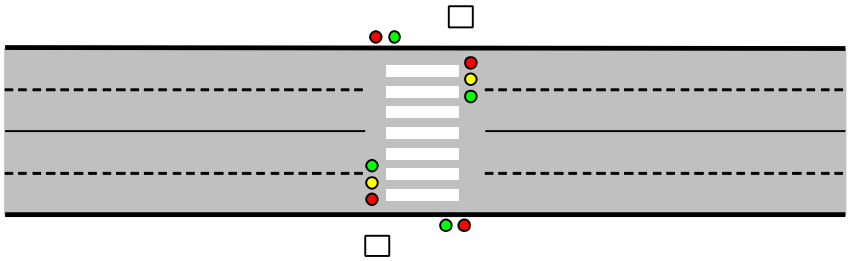


图 20　十字路口交通红绿灯示意图

图 20 是我们这节所要编写的交通红绿灯控制系统的应用场景，左右方向是主干道，上下方向是从干道，主干道行驶机动车辆，从干道为行人斑马线。主干道上的车多，通行时间长，从干道行人少，通行时间短，主从干道交替通行。顺着前进的方向看，主干道上的 3 个灯分别是红黄绿，从干道上的 2 个灯分别是红绿。上下的两个方块是行人横跨主干道时的应急按钮，当行人按下应急按钮时，无论主干道处于什么状态，主干道都会变为停止通行状态，从干道变为通行状态，行人可以从从干道通行，过一会主从干道又恢复为正常的交替通行状态。

表 3 描述了上述十字路口各个灯的状态运行情况：

	主干道红灯	主干道黄灯	主干道绿灯	从干道红灯	从干道绿灯
状态 1：主干道通行，从干道停止，30 秒。	灭	灭	亮	亮	灭
状态 2：主干道将停，从干道将通行，5 秒。	灭	亮	灭	亮	灭
状态 3：主干道停止，从干道通行，10 秒。	亮	灭	灭	灭	亮
状态 4：主干道将通行，从干道将停，5 秒。	亮	灭	灭	灭	闪烁

表 3　十字路口状态表

状态 1 持续 30 秒，主干道绿灯亮，从干道红灯亮，指示主干道通行从干道停止；此后转换为状态 2 持续 5 秒，主干道黄灯亮，从干道红灯亮，指示主干道将停止，从干道将通行；此后转换为状态 3 持续 10 秒，主干道红灯亮，从干道绿灯亮，指示主干道停止从干道通行；此后转换为状态 4 持续 5 秒，主干道红灯亮，从干道绿灯闪烁，指示主干道将通行，从干道将停止；此后再转换到状态 1，如此周而复始的运行。

另外，当行人按下应急按钮时，无论当前处于什么状态都会转换为从干道通行状态，此后仍按上述 4 个状态循环运行。

十字路口各个灯状态图转换如图 21 所示：

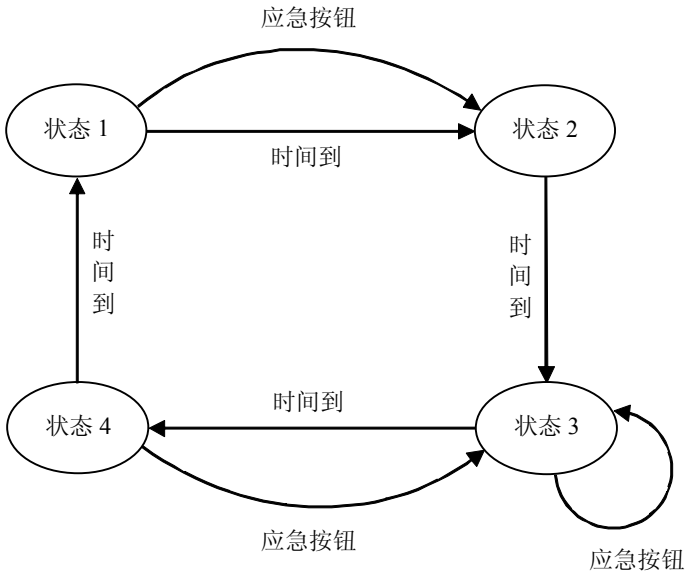


图 21　十字路口运行状态切换图

软件设计：

在任务设计上，需要尽可能做到任务间的耦合性小，任务之间仅通过少量的接口传递消息。在这个交通灯系统中，我们可以将其功能拆分成 2 任务，任务 1 用来控制十字路口的状态，并根据十字路口的状态改变各个灯的状态，任务 2 用来将各个灯的状态输出到灯上，这样分解的两个任务之间的耦合性小，当我们修改方案，需要改变十字路口各个灯的控制策略时，只需要修改任务 1 的代码，任务 2 几乎不受影响，这点在后面的例子中可以看到。

当行人按下应急按钮时会触发中断，在中断里面改变十字路口的状态，退出中断后，任务 1 会根据十字路口的状态重新更新各个灯的状态，任务 2 又会将灯的状态输出到灯上，这样就可以完成这个十字路口的软件功能了。

当然，也可以有其它的任务设计方式，但不管如何设计，必须要遵守的是：各个任务之间层次分明，耦合性要小，避免多个任务互相干扰。最差劲的设计是几个任务互相影响，比如一个任务控制主干道的灯，另一个任务控制从干道的灯，每个任务不仅要考虑到自己如何运行，还要考虑与另外一个任务的配合，当任务多的时候，这种配合将变的非常复杂，错误百出，即使功能实现了，这对于以后的维护、功能修改、扩展也将是巨大的考验。

在这个软件系统里，难点在于控制各个灯的状态变化。在所有的十字路口状态中，灯有亮、灭、闪烁 3 种状态，每种十字路口状态中每个灯有不同的持续时间，为此，我们可以用一个结构体来表示灯的这些状态：

```
typedef struct crossstatestr
{
    U32 uiRunTime;
    LEDSTATE astrLed[LEDNUM];
}CROSSSTATESTR;
```

其中 uiRunTime 是该状态运行的时间，LEDSTATE 结构体是每个灯的状态结构体，LEDNUM 是灯的数量，为每个灯定义一个状态变量。

LEDSTATE 结构体为：

```
typedef struct ledstate
{
    U32 uiLedState;
    U32 uiBrightness;
}LEDSTATE;
```

uiLedState 表示灯的状态，是亮、灭还是闪烁状态，uiBrightness 表示当灯处于闪烁状态时当前的亮度，是亮还是灭。

使用上面的这 2 个结构体就可以表示十字路口的一个状态。使用该结构体，我们定义一个十字路口的当前运行状态：

```
CROSSSTATESTR gstrCurCrossSta;
```

当十字路口状态发生变化时，需要重新获取新状态的各个参数，为此，我们再定义一个结构体数组用来存放十字路口的各个状态初始值：

```
CROSSSTATESTR gastrCrossSta[CROSSSTATENUM] = CROSSINITVALUE;
```

其中 CROSSSTATENUM 是十字路口状态的数量，这样变量 gastrCrossSta 中包含了所有灯的所有状态，CROSSINITVALUE 是变量 gastrCrossSta 的初始值，包含了表 3 中各个灯的所有状态，当程序运行时，各个灯的初始状态就全部被放到了 gastrCrossSta 变量中。

任务 1 运行时，若发现当前运行状态变量 gstrCurCrossSta 中的状态时间参数 uiRunTime 耗尽，gstrCurCrossSta 变量则从 gastrCrossSta 变量中获取下个状态的初始值，根据图 21 的状态切换关系改变状态，任务 2 再将灯的状态输出到灯上，如此循环。

在这个软件系统里，需要根据时间来改变各个灯的状态，我们可以使用硬件定时器每隔 100ms 产生一次中断，这个中断时间叫做 tick，是软件系统的时间单位，由软件在每个 tick 中断里对时间变量累计，这样在程序里对只要对时间变量进行判断就可以确定时间了。

软件流程如图 22、23 所示，软件开始运行时，初始化十字路口的各个状态，然后任务 1 和任务 2 交替运行。期间发生的定时器中断会更新时间变量计数，行人中断会改变十字路口的状态变量，任务 1 需要根据这些变量判断各个状态是否需要改变，如需改变时则发生状态切换，任务 2 再根据这些灯的状态更新灯的输出。注意一点，任务 1 在判断这些变量时需要锁中断，判断结束后再开中断，这样做是为了防止在判断状态变量时发生了中断，修改了这些变量，从而产生错误的判断。

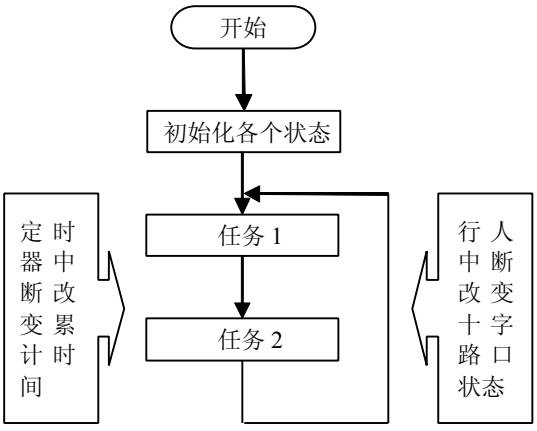


图 22　十字路口主流程图

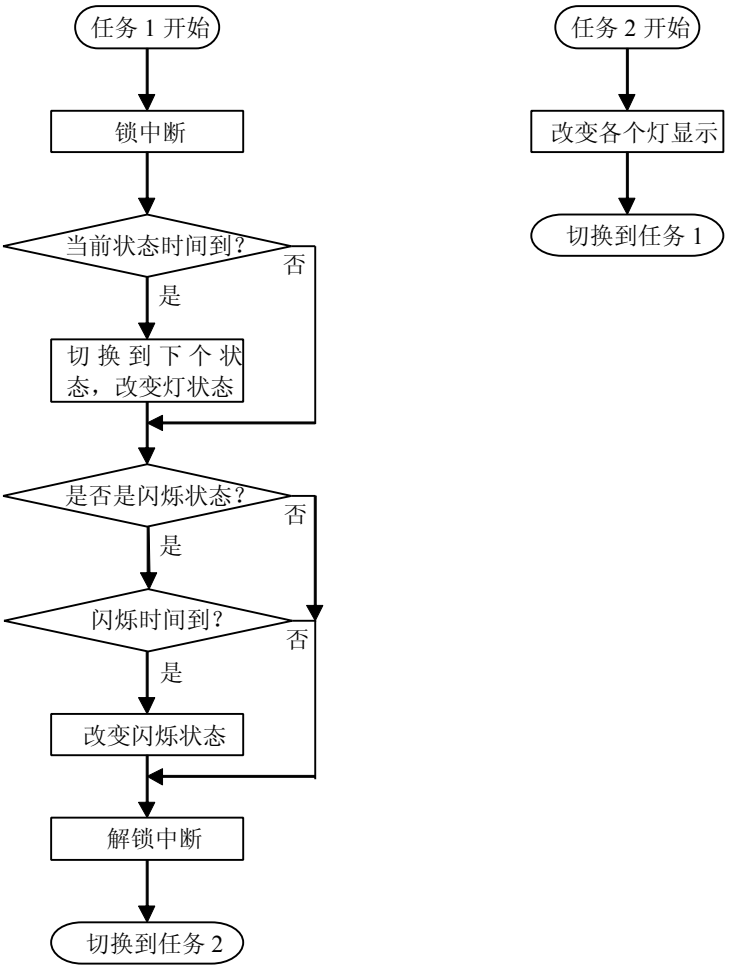


图 23　十字路口任务流程图

这个过程比较简单，就不详细介绍代码了。

下图是这个交通红绿灯控制系统的截图。由于我没有交通红绿灯单板，只能使用面包板搭建一个，简陋一些，但还是可以看到效果的。

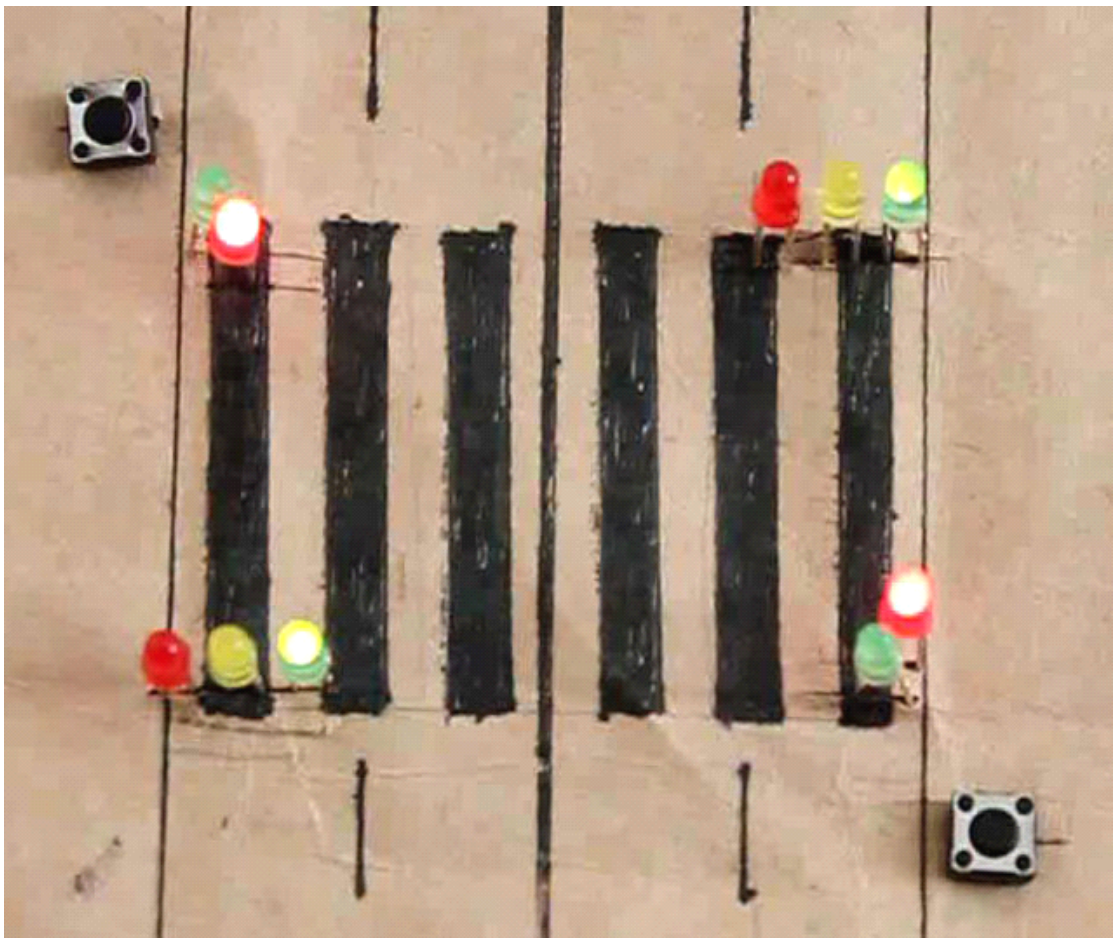


图 24　十字路口红绿灯演示

大家也可以到网站上去下载视频观看，可以看到各个灯按照我们的设计循环亮灭，当行人按下应急按钮时，可以中断主通道的通行，让行人先过马路，可以看到这个小系统实现了我们设计的要求。

最后，我们来做个小改动，在状态 1 和状态 2 之间增加一个状态：主干道绿灯闪烁，从干道红灯亮，用来表示主干道通行状态将要结束，姑且称之为状态 11，来看增加状态 11 后的表 4：

	主干道红灯	主干道黄灯	主干道绿灯	从干道红灯	从干道绿灯
状态 1：主干道通行，从干道停止，30 秒。	灭	灭	亮	亮	灭
状态 11：主干道通行，从干道停止，5 秒。	灭	灭	闪烁	亮	灭
状态 2：主干道将停，从干道将通行，5 秒。	灭	亮	灭	亮	灭
状态 3：主干道停止，从干道通行，10 秒。	亮	灭	灭	灭	亮
状态 4：主干道将通行，从干道将停，5 秒。	亮	灭	灭	灭	闪烁

表 4　增加状态后的十字路口状态表

增加了状态 11，我们在原有代码上需要做如下修改：

1.在 CROSSSTATENUM 枚举变量中增加一个新状态 11。

2.在 CROSSINITVALUE 宏定义中增加表 4 中状态 11 的初始值。

3.在行人中断函数 ISR_PassengerIsr 里增加对状态 11 的处理。

可以看到在这个软件结构上只要做很少的代码改动就增加这个新功能。大家切记，编码只是软件中的一部分工作，软件编码前的设计也非常重要！

第 5 节 发布 Wanlix 操作系统

经过前面 3.1~3.3 节循序渐进的开发，我们已经使 Wanlix 操作系统具备了最基本的任务切换功能，并在 3.4 节使用 Wanlix 开发了一个交通红绿灯控制系统，到此为止，我们已经完成了 Wanlix 操作系统的所有开发工作。Wanlix 操作系统的定位就是一个非常小巧的操作系统，有关嵌入式操作系统更多的功能，我们将在第 4 章开发 Mindows 操作系统的过程中不断的引入。

前面几节的代码里不仅包含了 Wanlix 的代码，而且还使用了一些用户代码来演示操作系统的功能，现在我们将操作系统的代码单独整理出来，去掉用户代码，发布仅有操作系统的代码，当用户需要使用这个操作系统时，只要在这些操作系统代码基础上补充自己的代码即可使用。

Wanlix 目录下的代码全部是操作系统代码，这个目录下的文件需要保留。srccode 目录下是用户代码，需要删除，但 wlx_userboot.c 和 wlx_userboot.h 文件作为操作系统与用户代码的接口文件需要保留，在 wlx_userboot.c 文件里需要清空 WLX_RootTask 函数里面的内容，wlx_userboot.h 文件里去掉对用户文件 global.h 的引用，去掉 system 目录下 Keil 开发环境使用的启动文件 startup.s 和链接文件 ADuC702X.ld，最后剩下的就仅仅是操作系统的代码了！

这个小操作系统虽然功能简单，但绝对可以实现任务的调度功能，这点对于一个小项目的程序设计来说就已经方便很多了，而且更难能可贵的是它耗费的系统资源是如此之少，编译后仅仅占用了 5、600 个字节的程序空间（不同编译器编译的结果会有所不同）和 16 个字节的内存空间。

当然，这个操作系统目前只能用在 ARM7 芯片上，因为任务切换的核心代码是用汇编编写的，而不同芯片的汇编语言又不兼容。因此，如果你需要在其它芯片上使用这个操作系统就必须修改 wlx_core_a.asm 文件里的函数，芯片的出入栈方式也是需要考虑的。

这个操作系统还有一个限制：任务不能运行到结束。不管是像 TEST_TestTask1 任务那样是个死循环，还是像 WLX_RootTask 任务那样执行一次之后就永远不会再切换回来继续执行了，一定要保证被创建任务的函数永远不能执行到最后一条指令。因为一个函数执行完后，会通过跳转指令返回到它的父函数，而 Wanlix 操作系统在创建任务时并没有为被创建任务的函数提供返回地址，因此被创建任务的函数就不能结束，这就是任务不能结束的原因。为了防止出现这种问题，每个任务需要使用 while 构造成一个死循环。本着 Wanlix 只实现最简单的任务切换功能的原则，这个问题在这里就不解决了，我们将在 Mindows 操作系统中解决。

另外，任务切换函数 WLX_TaskSwitch 不能在中断中使用。该函数会备份恢复任务的上下文信息，如果在中断中调用该函数则会破坏中断栈中的数据，导致系统崩溃。

仅剩下的这些操作系统代码只能编译，却不能链接出可执行的最终目标文件。

所谓编译就是将我们所写的 C、汇编等源代码翻译成芯片能理解的机器语言的过程，这

个过程中会使用一些技巧，减少冗余的代码，提高效率，这就是优化，例如

```
i++;
i++;
```

可以优化为：

```
i += 2;
```

当然，这个过程并非只是像这个例子这么简单，而是包含了大量的相当复杂的处理，以至于编译器的开发在相当长的一段时间内进展不大。在编译时有很多优化选项可以使用，Wanlix 所使用的 O2 选项就是其中的一个。

在程序出现的初期，是并没有编译这一概念的。那时候的电子设备没有键盘、显示器这样的输入输出设备，电子设备最小的功能单元就是一个个门电路，它只能识别 0 或者 1 的数字信号，因此，能想到的最方便的编程方法就是直接书写由 0、1 组成的程序，也就是直接写二进制机器码，类似 2.2 节中图 6 那样的格式，但这样编码不但效率低，而且极容易出错，修改、定位问题时更是痛苦万分。在编程过程中，人们逐渐发现，同一种操作都是对应同一个机器码，那么是否可以使用一个符号来代替这些机器码呢？这样就产生了汇编语言，例如在 ARM7 芯片上，使用汇编指令“ADD”来代替“加”操作的机器码“0b0100”，但这样替换后，也就需要一个翻译器将 ADD 符号及其后面的操作参数一起翻译成机器码，这就是编译器的概念。汇编语言的出现极大的提高了编程的效率，程序员们只需要记住汇编指令即可，不需要去查找复杂的机器码指令了。但汇编语言也只是简单的做了一些指令翻译的工作，程序运行的细节还需要程序员去关心，与硬件相关性也非常强，不同芯片的机器码也是不同的，汇编指令也是不同的，这给程序移植造成了很大的困难。这样编程语言就发展到了高级语言阶段，例如我们所使用的 C 语言。高级语言屏蔽掉了硬件层的概念，将程序语言抽象为接近人类逻辑的数字语言和自然语言，那么这个屏蔽硬件层、抽象语言的过程就完全由高级语言编译器来完成了。

源程序经过编译器的处理，被编译成了芯片可以识别的机器码，但此时还不能直接运行，因为编译过程只产生了机器码，并没有为这些机器码分配地址空间，前面我们介绍过，函数调用的过程就是 PC 指针跳转的过程，就是跳转到指令运行的地址空间取指的过程。每段程序必须有自己运行的空间，这是在链接过程中确定的，链接器会根据链接文件的配置，将已编译好的机器码分配到不同的地址空间，并计算各个函数、变量之间的地址关系，将他们关联起来，这样才会生成最终可执行的目标文件。

在链接过程，我们可以选择输出 map 文件，在 Keil 环境下可以选择“Project—>Options for target”，打开下面的对话框，选择“Listing”页，把红框内的“Memory Map”选上，重新编译链接就会生成 map 文件。

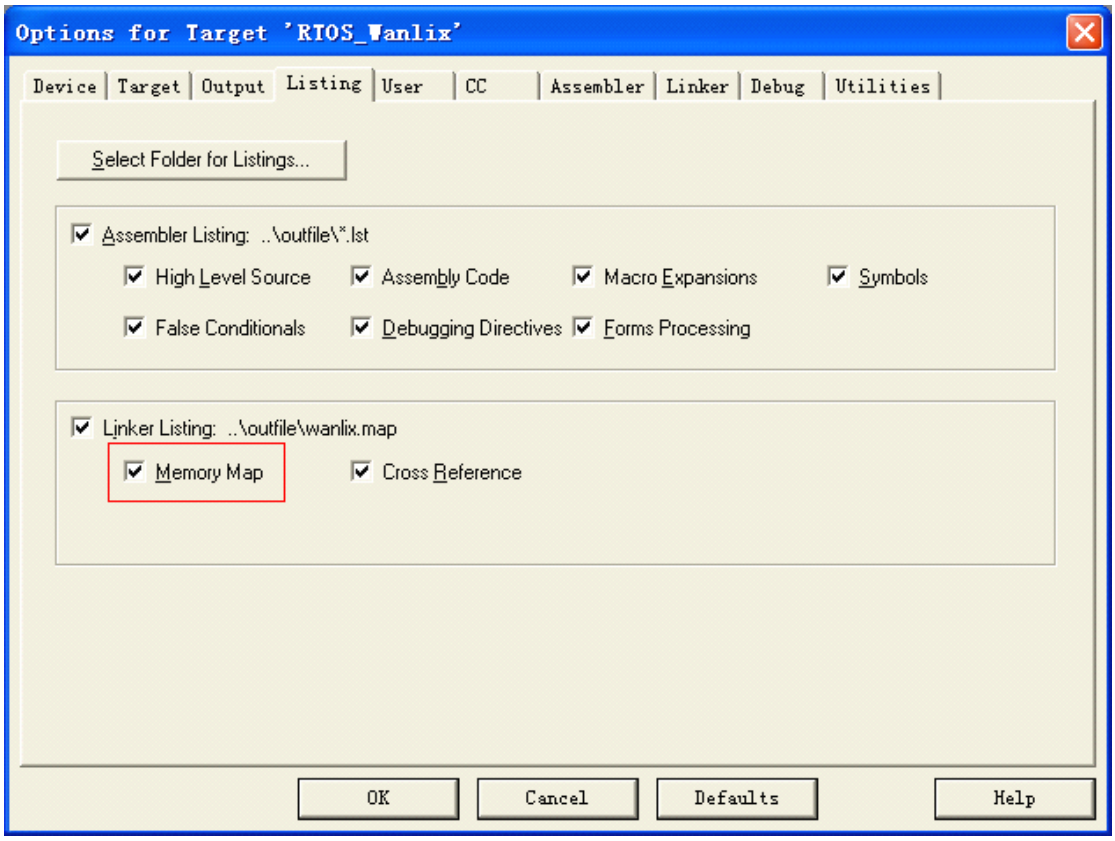


图 25　Keil 中生成 map 文件的选项

map 文件中包含了各个段、函数、全局等符号的地址分配情况，下面我截取了 3.4 节 map 文件中的一部分内容做个简单介绍：

```
006  Allocating common symbols
007  Common symbol       size              file
008
009  gaucRootTaskStack   0x190             ../outfile/wlx_core_c.o
010  gpstrRootTaskTcb    0x4               ../outfile/wlx_core_c.o
011  guiCurSta           0x4               ../outfile/test.o
```

……

```
025  Memory Configuration
026
027  Name             Origin             Length             Attributes
028  IntFLASH         0x00080000         0x0000f800         xr
029  IntRAM           0x00010000         0x00001b74         rw
030  *default*        0x00000000         0xffffffff
```

……

```
066   *(.text)
067   .text          0x00080108       0x54 ../outfile/wlx_core_a.o
068                  0x00080108             WLX_ContextSwitch
069                  0x0008013c             WLX_SwitchToTask
070   .text          0x0008015c      0x1f4 ../outfile/wlx_core_c.o
071                  0x0008015c             WLX_TaskTcbInit
```

……

227 0x00010308 guiCurSta

从 006 行可知 gaucRootTaskStack 全局变量的大小是 0x190 字节，在 wlx_core_c.c 文件中定义的。从 011 行可知 guiCurSta 全局变量的大小是 4 字节，在 test.c 文件中定义的，从 227 行可以知道它位于 0x00010308 的地址空间。从 028 行可知软件中有一个 IntFLASH 段，它从 0x00080000 地址开始，长度为 0x0000f800 字节，属性是只读和可执行。从 029 行可知软件中有一个 IntRAM 段，从 0x00010000 地址开始，长度为 0x00001b74 字节，属性是可读可写。从 067 和 068 行可知，WLX_ContextSwitch 函数位于 wlx_core_a.asm 文件中，它的起始地址是 0x00080108。

map 文件对软件开发还有会有一些帮助的，定位问题时我们可能会需要通过查找 map 文件来获得一些信息。不同工具生成的 map 文件格式是不同的，但内容大概都差不多，我这里只能抛砖引玉，遇到具体的情况还得读者自己分析。

本节发布的 Wanlix 代码，由于没有启动文件，也没有链接文件，因此不能生成可执行的目标文件，但我们可以将它们编译成库文件。在 Keil 环境下可以选择“Project—>Options for target”，打开下面的对话框，选择“Output”页，选择红框内的“Create Library”，重新编译就会生成库文件。

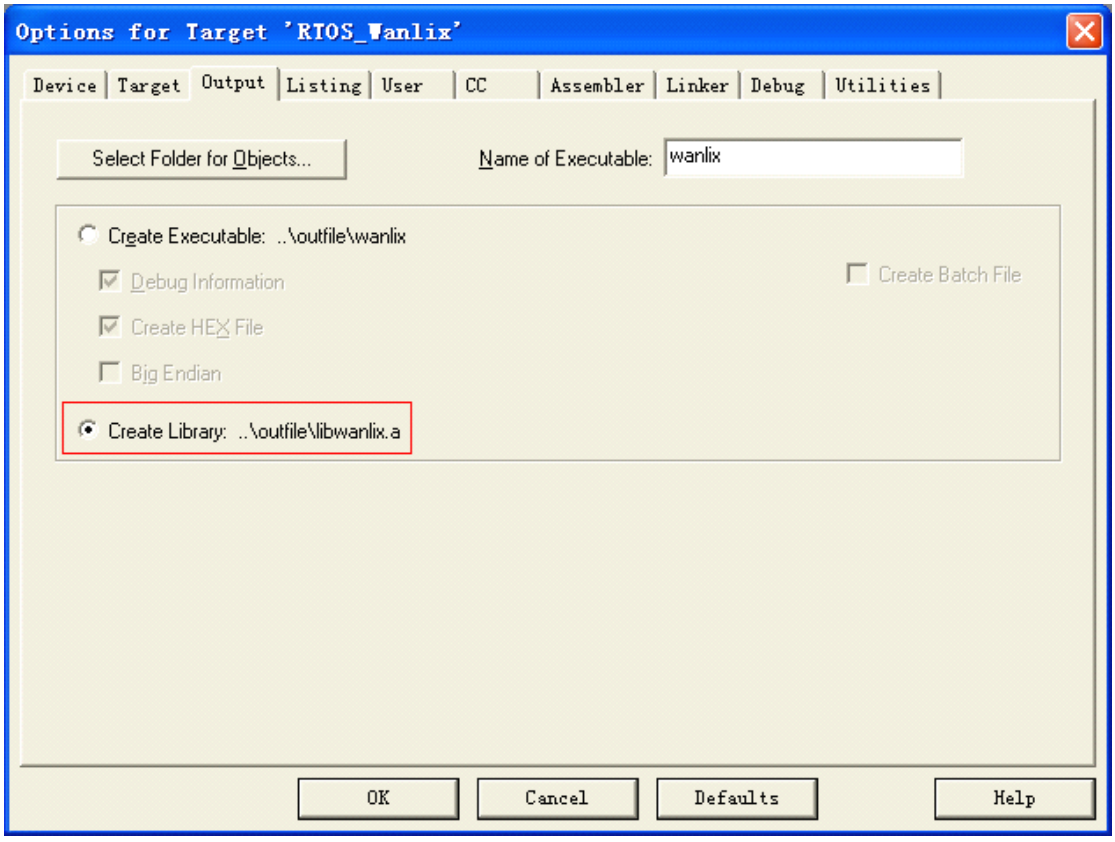


图 26 Keil 中生成库文件的选项

库文件是编译的结果，它里面包含了源代码编译后的机器码以及符号表，因此它可以作为链接过程的输入。我们可以将操作系统编译成库文件提供给用户使用，用户只要在自己的工程里包含库文件，就可以直接调用库文件里的函数、全局变量就可以了，而不需要拥有库文件的源代码。库文件的方式屏蔽了源代码，只提供机器码，对于不开源的软件，往往就是使用这种方法。对于某些较大的项目也可以使用这种方式开发，底层软件人员将他们的代码

编译成库文件，发布给上层软件人员使用，这样上层软件人员不需要全套代码就能编译出最终目标文件了。这样做不仅管理方便，而且也可以降低泄露产品全套源代码的风险。

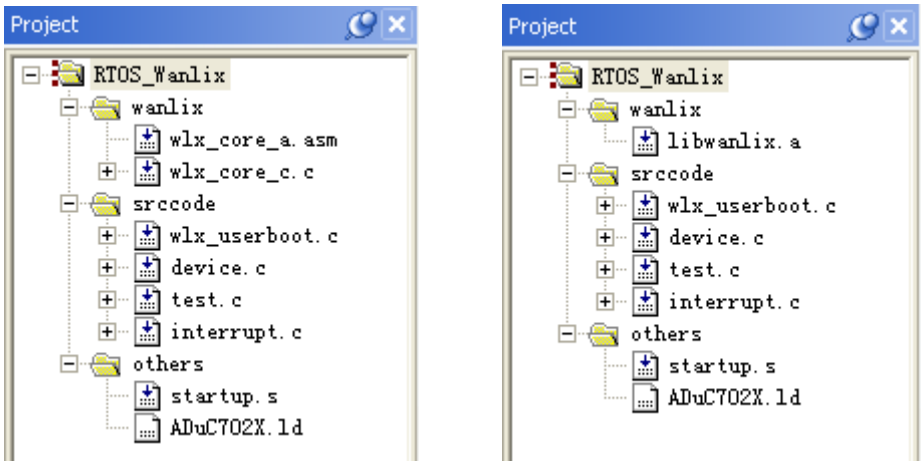


图 27　不使用库文件和使用库文件 Keil 工程对比

图 27 中，左侧是没有使用库文件的工程文件树结构，使用 Wanlix 的源代码编译。右侧是使用了库文件的工程文件树结构，使用 Wanlix 的库文件编译。注意，当使用库文件时，必须将库中的接口函数声明到一个 h 头文件中，使用库文件的程序也必须包含此头文件。在 Wanlix 操作系统中，这个头文件就是 wanlix.h，里面包含了 Wanlix 操作系统的全部对外接口函数，见附录 1。

写到这里，我遇到一个问题，在 GNU 环境下编译出的是.a 库文件，使用.a 库文件链接时链接程序出错了，不知道问题处在哪了，使用 Keil 自带的 RealView 编译链接器则没有此问题。如果哪位知道原因的话请到论坛上反馈一下，多谢了！

Wanlix 的开发到此就告一段落，我们最后为 Wanlix 设定一个版本号作为这一阶段的结束标志。版本号的格式为 Major.Minor.Revision.Build，Major 是主版本号，当软件功能或结构有重大改变时才修改此版本号，比如增加了多个重要功能或者整体架构发生变化。Minor 是子版本号，基于原有功能、结构增加、修改一些功能时修改此版本号。Revision 是修改版本号，当修改 bug、完善一些小功能时就更改此版本号。Build 是编译版本号，每次正式编译时该版本号加 1。每当上一级版本号变动后，下一级版本号归 0 重新开始。

Major 和 Minor 版本很重要，修改时需要产品相关人员参加讨论是否需要修改，如何修改，这 2 个版本往往影响了产品大的特性，对用户会有直接的影响。Revision 版本一般限于项目组内由项目经理控制，但需要发布给其它项目组使用。Build 版本一般限于项目组内部的修改测试，不对外公布。因此，产品发布新版本时，需要体现出 Major、Minor 和 Revision 版本，Build 版本建议不体现。

版本控制也非常重要，不能随意乱发导致版本过多，过多的版本会给产品维护带来非常多的麻烦，并且会增加维护成本。发布版本前需要做好版本计划，规划好一段时间内的版本数量，需要解决哪些问题，需要在哪个版本解决，版本发布后需要记录已发版本的特性、产品不同模块之间版本的配合使用关系等等问题。

版本号	说明			
	Major	Minor	Revision	Build
001.001.001.000	主版本号	子版本号	修正版本号	编译版本号

表 5 Wanlix 版本号格式

此次 Wanlix 发布的版本号为 001.001.001.000，只提供任务切换功能，Wanlix 后续若还有发展的话，就在此版本号基础上修改。

我最原始的计划只是将代码写到这里，写操作系统的初衷只是因为当时找不到一个适合小型嵌入式设备的操作系统，才萌生自己写一个只具有任务切换功能的操作系统的想法。但当我写到这里，实现了任务切换功能之后，我发现我还可以实现更多的功能，还可以讲述更多的原理，还可以让更多的人了解更多的操作系统知识，还可以继续写下去。因此，我将继续写下去，去编写一个功能更强大更完善的操作系统——Mindows 操作系统。

接下来的章节，我们将开始设计 Mindows 操作系统内核，一个具有实时抢占性的嵌入式操作系统内核！在编写 Mindows 的过程中我们将了解操作系统更多的知识，实现操作系统更多的功能！

第 4 章 Mindows 操作系统

在第 3 章中，我们实现了一个简单的操作系统——Wanlix，这个操作系统是一种非抢占式操作系统，任务之间的切换需要用户主动调用任务切换函数 WLX_TaskSwitch 来实现。从本章开始，我们将编写一个实时抢占的嵌入式操作系统——Mindows，它将具备操作系统更多的调度功能，在这个编写过程，我们将了解有关操作系统更多的内容。

第 1 节 Mindows 的文件组织结构

在第 3 章，我们依据任务切换的原理编写了 Wanlix 操作系统，Wanlix 很简单，只实现了任务切换功能，从本章开始，我们将编写 Mindows 操作系统，Mindows 将会实现更多的功能，相对 Wanlix 来说要复杂一些，文件也更多一些，因此，Mindows 在文件组织结构上做了一下调整，我们先来看一下 Mindows 的文件组织结构。

Mindows 的目录结构仍与 Wanlix 类似，有 mindows、srccode、others、project、outfile 这几个目录，mindows 目录下存放的是 Mindows 操作系统的文件，其余目录与 Wanlix 一致。

Mindows 操作系统的文件分为 2 类，一类是核心文件，操作系统需要依靠此类文件才可运行，例如 mds_core_c.c 文件，包含了操作系统的调度功能。另一类是非核心文件，操作系统脱离此类文件也可以运行，此类文件主要是提供操作系统的功能给用户使用。

核心文件包括：

mds_core_a.asm：操作系统的汇编文件，使用汇编语言编写的代码存放在此文件。

mds_core_c.c：使用 C 语言编写的操作系统核心调度代码存放在此文件。

mds_task.c：与任务相关的代码存放在此文件。

mds_sem.c：与信号量相关的代码存放在此文件。

mds_chip.c：与芯片强相关的代码存放在此文件，如果移植到不同芯片，需要修改较大的 C 函数代码存放在此文件完成。

mds_userboot.c：用户接口文件。

非核心文件包括：

mds_queue.c：与队列相关的代码存放在此文件。

mds_debug.c：与调试功能相关的代码存放在此文件。

以及以后可能会增加的文件。

操作系统的每个 c 文件有 2 个 h 头文件，比如说 mds_core_c.c 文件，它有 mds_core_c_inner.h 头文件和 mds_core_c.h 头文件。带“inner”的头文件是操作系统内部头文件，其中仅包含可在操作系统内部使用的信息，不提供给用户文件使用。不带“inner”的头文件是操作系统外部头文件，其中包含了可供操作系统和用户使用的信息，提供给用户文件使用。

除此之外，Mindows 还提供了 mds_mdsdef.h 和 mds_userdef.h 头文件，mds_mdsdef.h 里面包含了操作系统共用的一些信息，mds_userdef.h 里面的信息则是需要用户修改的，由用户根据项目的需要自己修改。

与 Wanlix 相似，Mindows 也提供一个对用户的接口头文件 mindows.h，mindows.h 是操

作系统的总头文件，它包含了所有核心 c 文件的外部头文件以及 mds_mdsdef.h 和 mds_userdef.h 头文件，用户需要包含 mindows.h 才可使用 Mindows 操作系统的功能。另外，Mindows 还提供了一个对内的总头文件 mindows_inner.h，它包含了所有核心 C 文件的内部头文件和 mindows.h 文件，并且它也被所有的操作系统 c 文件（mds_userboot.c 除外）所包含。

看了上面的说明会比较乱，我们来看看下面这个 Mindows 文件调用关系：

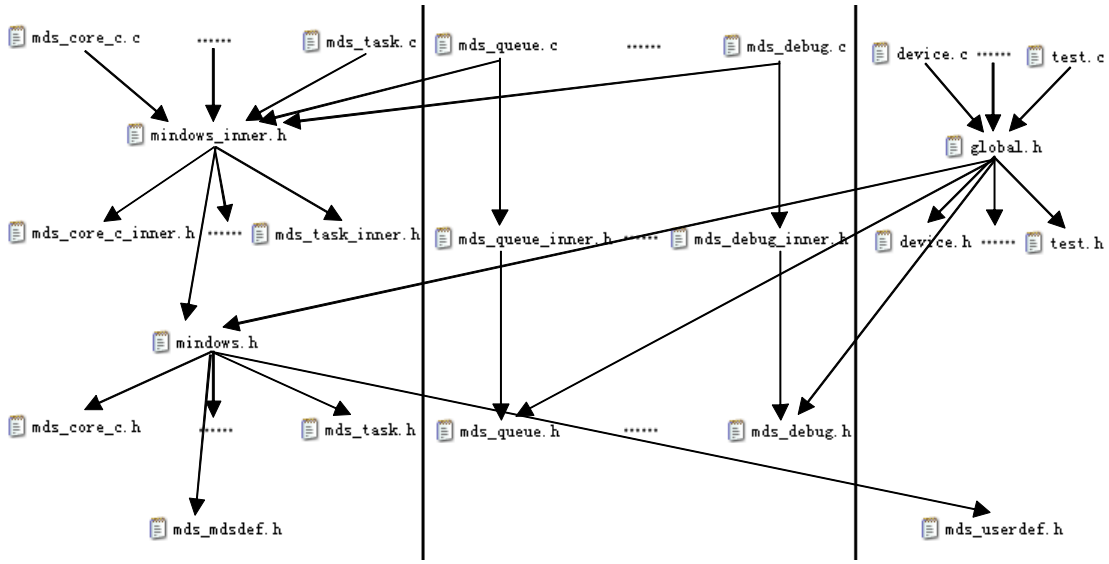


图 28 Mindows 文件调用关系

这么设计的初衷是为了将操作系统对内与对外使用的信息分开，将核心文件与非核心文件分开。将操作系统外部的信息都封装到 mindows.h 文件中，需要使用外部信息的文件只需要包含 mindows.h 文件即可。将操作系统内部和外部的信息都封装到 mindows_inner.h 文件中，操作系统文件只要包含 mindows_inner.h 文件即可。将核心文件都包含到 mindows.h 和 mindows_inner.h 文件中，需要使用核心文件时只需要包含 mindows.h 或 mindows_inner.h 文件即可，需要使用非核心文件时则需要单独包含非核心文件的头文件，这是因为，核心文件就那么几个，封装到 mindows.h 和 mindows_inner.h 文件中方便使用，而非核心文件则会随着操作系统功能的增加不断的扩充，而其中的信息可能也是与其它功能模块不相关的，因此，灵活的包含非核心模块则会更适合些。

本章的 Mindows 仍在 Aduc7024 芯片上使用 Keil 开发环境开发，使用 GNU 工具链，采用 O2 编译选项。

第 2 节 定时器触发的实时抢占调度

在第 3 章，我们依靠用户代码主动调用任务切换函数 WLX_TaskSwitch 实现了任务切换功能，这种任务调度方式的调度时机是固定死的，只有代码运行到 WLX_TaskSwitch 函数时才会发生任务切换，因此实时性不强。

实时操作系统采用 2 种调度方式确保它的实时性，一种是依靠硬件定时器触发的定时调度，另一种是依靠实时事件触发的随机调度，本节将讲述第一种调度方式。

实时操作系统会使用一个硬件定时器，利用硬件定时器定时产生中断，这个中断叫做 tick 中断，tick 中断的中断服务函数就是操作系统的任务调度函数，因此当设定好 tick 定时器后，实时操作系统就可以不依赖代码的运行情况，周期性的产生任务调度。操作系统的调度周期与 tick 周期有关，tick 周期越小操作系统的实时性越好，tick 周期越大操作系统的实时性越差，但 tick 周期也不是越小越好，因为产生 tick 中断时需要进行中断上下文切换和任务上下文切换，这个过程也是浪费时间的，如果 tick 周期过小，会使这一过程所占的比重过大，反而会降低芯片的效率，一般这个 tick 周期可以设置为 10ms，具体数值视系统整体情况而定。

从上面的介绍来看，实时操作系统也并非完全是实时的，那么它对比分时操作系统有什么优点呢？我们来看看 PC 机使用的 Windows 操作系统，它类似一种分时操作系统，Windows 在每个 tick 会调度一个任务，各个任务排队，在 2 个 tick 之间完成一个任务调度后便将这个任务放到队列的后面，下个 tick 将从队列前面的任务开始调度，就这样，每个任务都按照一定的顺序周而复始的运行。当在分时操作系统上同时运行很多任务时，这时候就可以明显的感觉到每个任务的执行速度变慢了，我们在 Windows 上同时运行很多大程序时就会体会到这种感觉。

而实时操作系统是不会出现这种情况的，实时操作系统中的任务是有优先级这个属性的，优先级越高的任务就越会被优先执行，优先级低的任务会等到优先级高的任务执行完毕再执行。依照优先级的调度方式，即使操作系统上同时运行了很多任务，但它会确保优先级高的任务先运行，当下个 tick 到来时，它还是运行优先级最高的任务，这样就确保了任务的实时性。

从上面的介绍可以看出，实时操作系统尽管保证了实时性，但它也是有代价的，它是以牺牲低优先级任务的实时性为代价来保证高优先级任务实时性的，不过，这也是合理的，物竞天择，重要的事情总是要优先处理的。当我们在实时操作系统上设计任务时，就需要根据任务功能的轻重缓急为任务分配适合的优先级，以保证整个系统的功能得以实现。

实时操作系统除了会优先执行高优先级的任务，还会发生高优先级任务抢占低优先级任务的情况。实时操作系统的任务可以分为 running、ready、delay、pend、suspend 等几种状态，任务在运行时可能会在这几种状态之间切换，在这个切换过程中就可能会发生任务抢占。

running：任务获取到 CPU 资源，任务正在运行。任何时刻，系统中只能有一个任务处于 running 状态。

ready：任务准备就绪，已经获取除 CPU 资源之外的所有资源。处于 ready 状态的任务如果具有最高优先级，那么下个任务调度就会执行它。

delay：任务主动延迟，放弃 CPU 资源。处于 running 状态的任务若暂时不需要执行，则可以调用相关函数进入 delay 状态，主动将 CPU 控制权交给其它任务。

pend：任务由于获取不到非 CPU 资源被阻塞。处于 running 状态的任务若获取不到非 CPU 资源，可能会被阻塞进入 pend 状态，被动将 CPU 控制权交给其它任务。

suspend：任务被挂起，不参与任务调度。这种状态一般只存在于调试过程中，在本手册中没有实现该状态，若读者感兴趣，可以自行编写代码实现。

这几种状态是可以相互转换的，如图 29 所示：

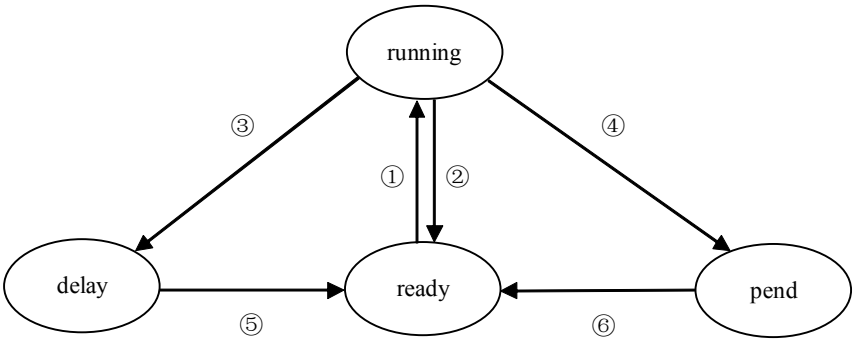


图 29　任务状态转换关系图

①ready—>running：任务获取到所有资源，包括 CPU 资源，正在运行。

②running—>ready：任务被其它任务抢占，丧失 CPU 资源。

③running—>delay：任务主动调用 MDS_TaskDelay 函数，任务被延迟，丧失 CPU 资源。

④running—>pend：任务使用 MDS_SemTake 等函数获取不到资源，任务被阻塞，丧失 CPU 资源。

⑤delay—>ready：任务延迟时间耗尽或者被 MDS_TaskWake 函数唤醒，等待 CPU 资源。

⑥pend—>ready：任务阻塞时间耗尽或者任务获取到了被阻塞的资源，等待 CPU 资源。

从图 29 中可以看出，任务如果要转换成 running 态去运行，必须得先转换成 ready 态，操作系统在任务调度时只会从处于 ready 态的任务中查找最高优先级的任务去执行，至于那些即便是拥有了最高优先级的任务，只要它不是处于 ready 态，任务调度时也不会考虑，现在我们就来设计这个 ready 态。

同一时刻会有很多任务处于 ready 态，这些处于 ready 态的任务的优先级是一个非常重要的属性，操作系统就是依靠优先级这个属性来调度任务的。我们可以做一个 ready 表，使用这个 ready 表将各个不同优先级的任务关联起来，任务调度时只需要查找 ready 表就可以找到各个任务了。我们先以 8 级（0~7）优先级为例，如图 30 所示：

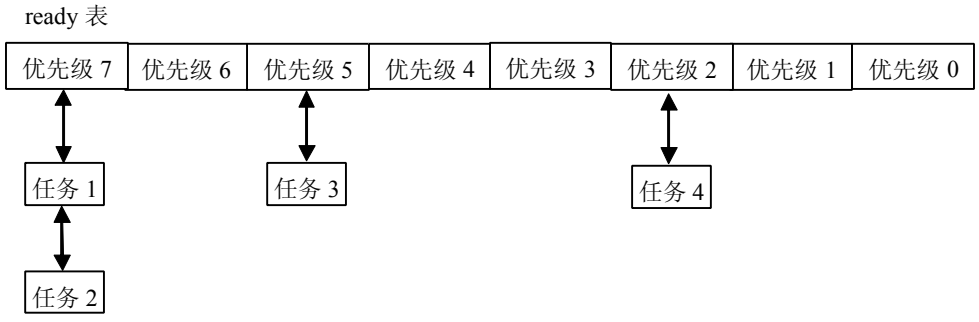


图 30　ready 表与任务的关联关系图

任务 1 和任务 2 的优先级都是 7，与 ready 表的“优先级 7”相关联，任务 3 的优先级是 5，与 ready 表的“优先级 5”相关联，任务 4 的优先级是 2，与 ready 表的“优先级 2”相关联。与 ready 表关联的任务是可以改变的，可以增加也可以减少，我们可以采用链表来关联 ready 表与各个任务，ready 表中的每个“优先级”就是这个优先级链表的根节点，与之相关的任务就是这个链表上的一个节点。比如“优先级 7”就是 ready 表中优先级为 7 的链表的根节点，任务 1 和任务 2 分别是优先级为 7 的链表中的一个节点。这样当我们需要查

找 ready 表中优先级为 7 的任务时，只需要先找到优先级为 7 的链表根节点，再从根节点顺着链表就可以找到优先级为 7 的任务有任务 1 和任务 2。

我们来看看 Mindows 里使用的链表结构体：

```
typedef struct m_chain              /* 链表结构 */
{
    struct m_chain* pstrHead;       /* 头指针 */
    struct m_chain* pstrTail;       /* 尾指针 */
}M_CHAIN;
```

pstrHead 是链表的头指针，pstrTail 是链表的尾指针。当链表为空，即链表只有根节点时，根节点的头尾指针都指向空指针。当链表不为空时，根节点的头指针指向链表中的第一个子节点，根节点的尾指针指向链表中的最后一个子节点，每个子节点的头指针指向它前面的节点（包括根节点），每个子节点的尾指针指向它后面的节点（包括根节点）。这样所有的子节点与根节点就组成了一个环状的双向链表结构，通过根节点可以快速找到链表的第一个和最后一个子节点，通过任何一个节点都可以找到它前后的节点。我们来看看链表为空，有 1 个节点，2 个节点和多个节点的情况：

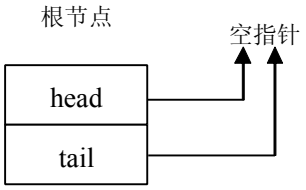


图 31　空链表

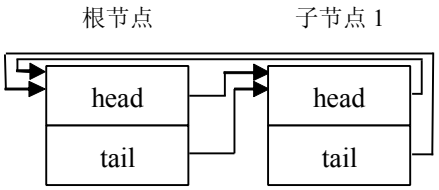


图 32　拥有 1 个子节点的链表

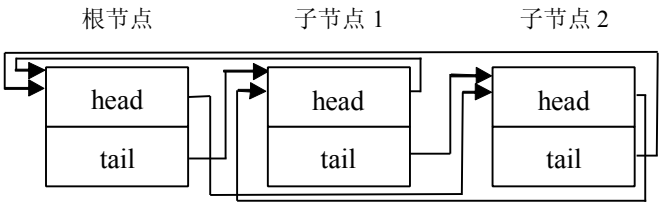


图 33　拥有 2 个子节点的链表

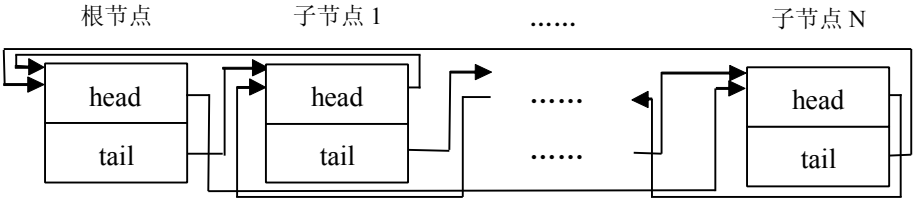


图 34　拥有多个子节点的链表

Mindows 提供了几个链表操作函数，包括：初始化链表的函数 MDS_ChainInit，向链表尾部插入一个节点的函数 MDS_ChainNodeAdd，从链表头部删除一个节点的函数 MDS_ChainNodeDelete，向链表指定的节点前插入一个节点的函数 MDS_ChainCurNodeInsert，删除链表中指定的节点的函数 MDS_ChainCurNodeDelete，查询链表是否为空的函数 MDS_ChainEmpInq，查询链表中指定节点的下一个节点是否为空的函数 MDS_ChainNextNodeEmpInq。这些链表操作函数将会在 Mindows 中用到，链表操作函数的细节这里就不具体介绍了，请读者自己参考代码。

拥有了链表结构，ready 表就可以用链表数组来实现，如下定义了具有 8 个元素的链表数组 astrChain[8]，每个数组元素分别对应 ready 表里每个“优先级”链表的根节点，astrChain[0]~astrChain[7]分别对应优先级 0~优先级 7 链表的根节点。

```
M_CHAIN astrChain[8];
```

任务若要挂到 ready 表上，那么任务里也需要有一个 M_CHAIN 的链表节点结构，任务的这个链表节点结构就被放到了 TCB 里面，后面我们在介绍 TCB 的时候再详细说。

根据前面链表的定义我们知道，当链表为空时，链表根节点的头尾指针指向 NULL，我们可以根据这一点来查找 ready 表中所挂接的最高优先级任务。但这样做有一个问题，当 ready 表中有最高优先级为 0 的任务时（在 Mindows 中，我们定义优先级 0 为最高优先级），我们只需要查找 astrChain[0]的链表，发现 astrChain[0]的链表不为空就不需要再继续查找了，而当 ready 表中只有优先级为 7 的任务时，我们则需要依次查询 strChain[0]~strChain[7]的链表，这样做不但效率不高，而且查询 ready 表所花费的时间也不固定。实时操作系统不但需要有实时性，而且调度时间也需要有确定性，这样系统才能运行平稳，调度时间不会有大的波动。为了弥补这个缺陷，我们为 ready 表的每个根节点配备一个标志，用这些标志来指明这些根节点是否为空，这样，我们就可以通过这些标志快速的查找到挂有任务的最高优先级的链表根节点，而不需要直接去查询每个根节点里的头尾指针是否为 NULL，如图 35 所示：

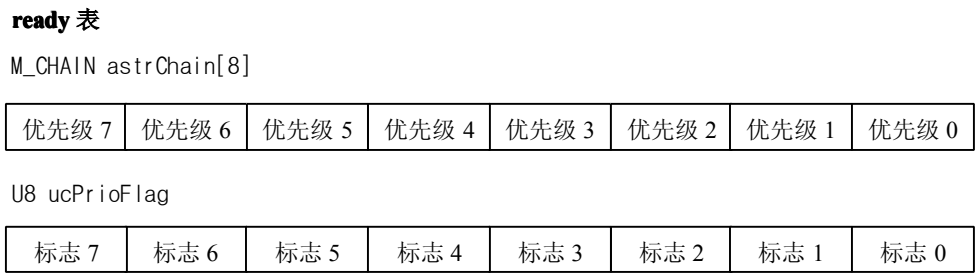


图 35 ready 表链表根节点与标志的对应关系图

8 个标志位对应 8 个优先级的根节点，我们正好可以定义 1 个字节的变量 ucPrioFlag，使用这个变量里面的每个 bit 作为一个标志来表示各个优先级的链表是否为空，bit0~bit7 分别对应 astrChain[0]~astrChain[7]，bit 为 0 代表链表为空，bit 为 1 代表链表不为空，这个优先级的链表上有任务处于 ready 状态。

到这里，我们已经完成了具有 8 个优先级 ready 表的设计，我们再来看看对 ready 表的操作过程。

◆ 初始化 ready 表

将 ready 表中的每个根节点的头尾指针都指向 NULL，使 8 个优先级的链表都为空，同

时，将标志置为 0，这时候 ready 表上没有挂接任何任务。

◆ 将节点添加到 ready 表

当有任务变为 ready 态时，我们将任务 TCB 中的链表节点添加到 ready 中相同优先级的链表上，并将对应的标志 bit 置为 1。

◆ 从 ready 表拆离一个节点

当有任务需要从 ready 表拆离时，我们就将这个任务的节点从 ready 表中相同优先级的链表中拆除，同时将对应的标志 bit 置为 0。

◆ 查询 ready 表中的最高优先级

这个过程稍微复杂一些，听我慢慢道来。

我们来看看 ready 表标志为下面几种情况的例子：

bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	bit0	最高优先级
1	0	0	0	0	0	0	0	7
1	1	0	0	0	0	0	0	6
1	1	1	0	0	0	0	0	5
1	0	1	0	0	0	0	0	5
0	1	1	0	1	0	1	1	0
1	0	1	0	0	0	0	1	0

表 6　ready 表标志与优先级关系

通过这几组数据我们可以看出来，ready 表中的最高优先级是由标志中为 1 的最低 bit 决定的，也就是说，从 bit0 向 bit7 找，当发现第一个为 1 的 bit 时，这个 bit 对应的就是任务的最高优先级，与后面的高位 bit 无关。8bits 共有 256 种组合，我们可以将这 256 种组合一一列出：

当标志为 0b000000001 的时候，也就是 1 的时候，优先级为 0
当标志为 0b000000010 的时候，也就是 2 的时候，优先级为 1
当标志为 0b000000011 的时候，也就是 3 的时候，优先级为 0
当标志为 0b000000100 的时候，也就是 4 的时候，优先级为 2
当标志为 0b000000101 的时候，也就是 5 的时候，优先级为 0
……
当标志为 0b11111110 的时候，也就是 254 的时候，优先级为 1
当标志为 0b11111111 的时候，也就是 255 的时候，优先级为 0

如果我们将“标志的值”作为数组下标，将“优先级”作为数组元素值的话，我们就可以构造出下面的这个数组：

```
const U8 caucTaskPrioUnmapTab[256] =    /* 优先级反向查找表 */
{
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
```

```
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

当我们需要查找 ready 表中的最高优先级时，就可以把标志 ucPrioFlag 当做数组下标带入到数组 caucTaskPrioUnmapTab 中，数组元素的值 caucTaskPrioUnmapTab [ucPrioFlag]就是 ready 表中的最高任务优先级。这样查找最高优先级的过程非常简单，速度快且花费时间相同，而且系统开销也不大。

当标志值为 0 时，这种情况是不存在的，ready 表中至少会有一个任务，至少处于 running 态的任务就位于 ready 表中，所以 caucTaskPrioUnmapTab[0]的值是无效的，只是用来占个坑填满组数。

8 个优先级的 ready 表相关内容已经全部讲述完毕。

8 个优先级对于小型嵌入式设备来说应该是够用了，但对稍大一些的嵌入式设备来说就显得太少了。我们可以通过扩充 astrChain 数组的数组元素数量来扩充 Mindows 所支持的优先级数量，同时，也需要相应的扩充 ucPrioFlag 标志的长度好与之相对应。通过这种方法，我们可以将 Mindows 支持的优先级数量扩展到支持 8、16、32、64、128、256 这 6 种不同的级别上，用下面的宏分别区分这些级别：

```
#define PRIORITY256         256
#define PRIORITY128         128
#define PRIORITY64          64
#define PRIORITY32          32
#define PRIORITY16          16
#define PRIORITY8           8
```

用户在确定使用哪个数量的优先级时，需要在 mds_userdef.h 文件里将 PRIORITYNUM 宏定义为上面这 6 种优先级数量中的一个，例如，用户使用下面的宏定义，选择 Mindows 支持 32 个优先级数量：

```
#define PRIORITYNUM         PRIORITY32
```

操作系统通过 PRIORITYNUM 宏就可以知道用户所希望使用的优先级数量了，那么，现在我们把 ready 表的结构重新整理一下：

```
typedef struct m_taskschedtab           /* 任务调度表 */
{
    M_CHAIN astrChain[PRIORITYNUM];     /* 各个优先级根节点 */
    M_PRIOFLAG strFlag;                 /* 优先级标志 */
}M_TASKSCHEDTAB;
```

astrChain 仍是链表数组，每个数组元素对应一个优先级的链表根节点，优先级的数量由 PRIORITYNUM 宏确定。strFlag 是每个优先级链表对应的标志，M_PRIOFLAG 结构体如下：

```
typedef struct m_prioflag               /* 优先级标志表 */
{
#if PRIORITYNUM >= PRIORITY128
    U8 aucPrioFlagGrp1[PRIOFLAGGRP1];
    U8 aucPrioFlagGrp2[PRIOFLAGGRP2];
```

```
    U8 ucPrioFlagGrp3;
#elif PRIORITYNUM >= PRIORITY16
    U8 aucPrioFlagGrp1[PRIOFLAGGRP1];
    U8 ucPrioFlagGrp2;
#else
    U8 ucPrioFlagGrp1;
#endif
}M_PRIOFLAG;
```

M_PRIOFLAG 结构体根据不同优先级数量将标志位分为 3 种情况：只定义 8 个优先级数量的时候标志位只使用 1 个字节就可以了，定义 16、32、64 个优先级数量的时候标志位分为了 2 级，定义 128、256 个优先级数量的时候标志位分为了 3 级，下面我们来看看为什么要这么做。

在优先级只有 8 个的时候，标志有 2^8 共 256 种组合，我们可以使用一个 256Bytes 的数组 caucTaskPrioUnmapTab 来构建优先级反向查找表。当优先级为 16 个的时候，标志有 2^{16} 共 65536 种组合，若构建 64KBytes 的反向查找表未免显得太浪费了，而且对于某些小型嵌入式系统来说也无法实现。当优先级为 256 个时，2^{256} 是一个非常巨大的数，任何硬件都无法支持这么大的数组。

为了解决这个问题，我们可以将 ready 表的标志分级，以 256 个优先级为例，如图 36 所示：

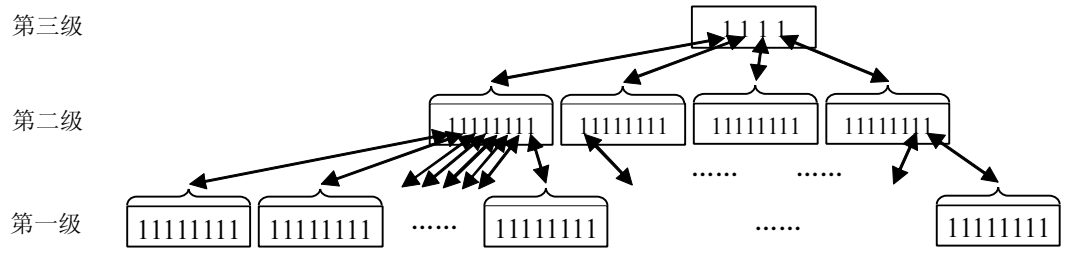


图 36　ready 表 256 级标志分级方法

优先级反向查找表只支持 8 个优先级，256 可以分解为 4×8×8，第三级有 4 个 bits，每个 bit 分别对应第二级的一个 Byte，第二级每个 Byte 中的每个 bit 分别对应第一级的一个 Byte，这样到第一级就有 4×8×8 共 256 个 bits，每个 bit 对应一个优先级链表，用来指示该优先级链表是否为空。第一级标志对应着每个优先级链表，第二级和第三级是为了查找第一级而设立的，我们来看看操作 ready 表时这个 3 级标志是怎么处理的：

◆ 初始化 ready 表

将 3 级标志全部置为 0。

◆ 将节点添加到 ready 表

将每一级标志中对应的 bit 置为 1。比如说添加的任务优先级是 143，使用整型数计算舍弃小数，143÷8÷8=2，对应第三级的 bit2，将第三级的 bit2 置为 1。143÷8=17，对应第二级的 bit17，17÷8=2，对应第二级的 Byte2，17－8×2=1，对应 Byte2 的 bit1，将第二级 Byte2 中的 bit1 置为 1。143÷8=17，对应第一级的 Byte17，143－8×17=7，对应 Byte17 的 bit7，将第一级 Byte17 中的 bit7 置为 1。

◆ 从 ready 表拆离一个节点

将第一级标志中对应的 bit 置为 0，若该 bit 所属的第二级和第三级中没有其它任务的标志，则将第二级、第三级对应的 bit 也置为 0，否则维持原状不变。比如 ready 表中有 143 和 144 这两个优先级的任务，如果要拆除 143 时，将第一级 143 的标志置为 0，而第二级 Byte2 中的 bit1 和第三级的 bit2 是不能置为 0 的，因为 144 优先级也位于第二级 Byte2 中的

bit1 和第三级的 bit2。

◆ 查询 ready 表中的最高优先级

查询 3 级标志的优先级时仍使用 caucTaskPrioUnmapTab 这个优先级反向查询表，只不过是分了 4 步。第 1 步，从第三级 4bits 里查询出第二级中拥有最高优先级的 Byte，第 2 步，从第二级的这个 Byte 里查询出第一级中拥有最高优先级的 Byte，第 3 步，从第一级的这个 Byte 里查询出拥有最高优先级的 bit，第 4 步，通过前 3 步找出的最高优先级所在第一级、第二级和第三级中所在的位置，算出最高优先级。

我们仍以 143 优先级为例，通过上面的讲述，我们知道 143 优先级在第三级的 bit2，在第二级 Byte2 中的 bit1，在第一级 Byte17 中的 bit7。第 1 步，第三级标志为 0b0100，查询优先级反向表 caucTaskPrioUnmapTab[4]的值为 2，说明最高优先级在第二级的 Byte2 中。第 2 步，Byte2 的值为 0b00000010，查询优先级反向表 caucTaskPrioUnmapTab[2]的值为 1，说明最高优先级在第二级 Byte2 中所拥有的 8 个第一级 Bytes 中的 Byte1 中，8×2＋1=17，也就是第一级中的 Byte17。第 3 步，Byte17 的值为 0b10000000，查询优先级反向表 caucTaskPrioUnmapTab[128]的值为 7，说明最高优先级在第一级 Byte17 中的 bit7。第 4 步，(8×2＋1)×8＋7=143，这样我们就查找到了 ready 表中的最高优先级。

查找到最高优先级后，astrChain[最高优先级]就是最高优先级链表的根节点，通过此链表就可以找到最高优先级任务的节点，而最高优先级任务的节点与最高优先级任务的 TCB 相关联，因此就可以找到最高优先级任务的信息了。关于本节的任务 TCB 结构，我们在后面再讲述。

本节主要是增加了 ready 调度表，原理讲解到这里，下面来看操作 ready 表的代码。

MDS_TaskAddToSchedTab 函数的作用是将任务添加到 ready 表中：

```
00194  void MDS_TaskAddToSchedTab(M_CHAIN* pstrChain, M_CHAIN* pstrNode,
00195                             M_PRIOFLAG* pstrPrioFlag, U8 ucTaskPrio)
00196  {
00197      /* 将该任务节点添加到调度表中 */
00198      MDS_ChainNodeAdd(pstrChain, pstrNode);
00199
00200      /* 设置该任务对应调度表的优先级标志表 */
00201      MDS_TaskPrioFlagSet(pstrPrioFlag, ucTaskPrio);
00202  }
```

194 行，入口参数 pstrChain 是 ready 表中优先级链表的根节点指针，也就是 astrChain 数组里面的一个数组元素的指针。入口参数 pstrNode 是需要挂接到链表上的节点。入口参数 pstrPrioFlag 是优先级标志的指针。入口参数 ucTaskPrio 是挂入的任务的优先级。

这个函数的作用就是将优先级为 ucTaskPrio 的任务的节点 pstrNode，挂入到 pstrChain 链表上，同时将 pstrPrioFlag 标志中对应的位置置为 1。

198 行，将子节点添加到链表中。

201 行，设置对应位置的标志。

MDS_TaskPrioFlagSet 函数的原理前面已经介绍过，代码如下，就不一一讲解了。

```
00230  void MDS_TaskPrioFlagSet(M_PRIOFLAG* pstrPrioFlag, U8 ucTaskPrio)
00231  {
00232  #if PRIORITYNUM >= PRIORITY128
00233      U8 ucPrioFlagGrp1;
00234      U8 ucPrioFlagGrp2;
```

```
00235     U8 ucPosInGrp1;
00236     U8 ucPosInGrp2;
00237     U8 ucPosInGrp3;
00238 #elif PRIORITYNUM >= PRIORITY16
00239     U8 ucPrioFlagGrp1;
00240     U8 ucPosInGrp1;
00241     U8 ucPosInGrp2;
00242 #endif
00243
00244     /* 设置调度表对应的优先级标志表 */
00245 #if PRIORITYNUM >= PRIORITY128
00246
00247     /* 获取优先级标志在第一组和第二组中的组号 */
00248     ucPrioFlagGrp1 = ucTaskPrio / 8;
00249     ucPrioFlagGrp2 = ucPrioFlagGrp1 / 8;
00250
00251     /* 获取优先级标志在每一组中的位置 */
00252     ucPosInGrp1 = ucTaskPrio % 8;
00253     ucPosInGrp2 = ucPrioFlagGrp1 % 8;
00254     ucPosInGrp3 = ucPrioFlagGrp2;
00255
00256     /* 在每一组中设置优先级标志 */
00257     pstrPrioFlag->aucPrioFlagGrp1[ucPrioFlagGrp1] |= (U8)(1 << ucPosInGrp1);
00258     pstrPrioFlag->aucPrioFlagGrp2[ucPrioFlagGrp2] |= (U8)(1 << ucPosInGrp2);
00259     pstrPrioFlag->ucPrioFlagGrp3 |= (U8)(1 << ucPosInGrp3);
00260
00261 #elif PRIORITYNUM >= PRIORITY16
00262
00263     ucPrioFlagGrp1 = ucTaskPrio / 8;
00264
00265     ucPosInGrp1 = ucTaskPrio % 8;
00266     ucPosInGrp2 = ucPrioFlagGrp1;
00267
00268     pstrPrioFlag->aucPrioFlagGrp1[ucPrioFlagGrp1] |= (U8)(1 << ucPosInGrp1);
00269     pstrPrioFlag->ucPrioFlagGrp2 |= (U8)(1 << ucPosInGrp2);
00270
00271 #else
00272
00273     pstrPrioFlag->ucPrioFlagGrp1 |= (U8)(1 << ucTaskPrio);
00274
00275 #endif
00276 }
```

MDS_TaskHighestPrioGet 函数是查询 ready 表中最高优先级的函数，入口参数 pstrPrioFlag 是优先级标志的指针，原理前面已经介绍过，代码如下，不再做介绍了。

```
00283 U8 MDS_TaskHighestPrioGet(M_PRIOFLAG* pstrPrioFlag)
00284 {
00285 #if PRIORITYNUM >= PRIORITY128
00286     U8 ucPrioFlagGrp1;
00287     U8 ucPrioFlagGrp2;
00288     U8 ucHighestFlagInGrp1;
00289 #elif PRIORITYNUM >= PRIORITY16
00290     U8 ucPrioFlagGrp1;
00291     U8 ucHighestFlagInGrp1;
00292 #endif
00293
00294     /* 获取任务调度表中的最高优先级 */
```

```
00295  #if PRIORITYNUM >= PRIORITY128
00296
00297      ucPrioFlagGrp2 = caucTaskPrioUnmapTab[pstrPrioFlag->ucPrioFlagGrp3];
00298
00299      ucPrioFlagGrp1 =
00300                  caucTaskPrioUnmapTab[pstrPrioFlag->aucPrioFlagGrp2[ucPrioFlagGrp2]];
00301
00302      ucHighestFlagInGrp1 = caucTaskPrioUnmapTab[pstrPrioFlag->aucPrioFlagGrp1
00303                                                 [ucPrioFlagGrp2 * 8 + ucPrioFlagGrp1]];
00304
00305      return (U8)((ucPrioFlagGrp2 * 8 + ucPrioFlagGrp1) * 8 + ucHighestFlagInGrp1);
00306
00307  #elif PRIORITYNUM >= PRIORITY16
00308
00309      ucPrioFlagGrp1 = caucTaskPrioUnmapTab[pstrPrioFlag->ucPrioFlagGrp2];
00310
00311      ucHighestFlagInGrp1 =
00312                  caucTaskPrioUnmapTab[pstrPrioFlag->aucPrioFlagGrp1[ucPrioFlagGrp1]];
00313
00314      return (U8)(ucPrioFlagGrp1 * 8 + ucHighestFlagInGrp1);
00315
00316  #else
00317
00318      return caucTaskPrioUnmapTab[pstrPrioFlag->ucPrioFlagGrp1];
00319
00320  #endif
00321  }
```

操作系统支持的优先级数量越多，占用的系统资源也就越多，不但对 ready 表操作的时间会增加，而且还会占用内存空间，如表 7 所示：

优先级数量	根节点字节数	一级标志字节数	二级标志字节数	三级标志字节数	总字节数
8	64	1	0	0	65
16	128	2	1	0	131
32	256	4	1	0	261
64	512	8	1	0	521
128	1024	16	2	1	1043
256	2048	32	4	1	2085

表 7　优先级数量与需要使用的内存数量

从表 7 可以看到，8 个优先级的 ready 表只需要使用 65 个字节就可以了，这对资源少的嵌入式系统来说是完全可以接受的，而 256 个优先级的 ready 则需要使用 2085 个字节，单单一个 ready 表就已经超过了 2KBytes 的内存，这对只有几 KBytes 或者更少内存的嵌入式设备来说已经不适合了。但 2KBytes 的内存对于拥有几百 MBytes 甚至几 GBytes 的嵌入式设备来说却又不算什么，这样的大系统也会很复杂，8 个优先级远远不能满足需求，也需要操作系统能支持 256 个优先级。我们在设计软件系统时，需要根据硬件资源的限制，合理选择 Mindows 所支持的优先级数量。

ready 表就介绍到这里，接下来我们来看一下 TCB 结构。

```
typedef struct m_tcb
{
    STACKREG strStackReg;           /* 备份寄存器组 */
    M_TCBQUE strTcbQue;             /* TCB 结构队列 */
    U8 ucTaskPrio;                  /* 任务优先级 */
```

```
}M_TCB;
```

TCB 里面有寄存器组、一个队列、还有任务的优先级。

在 Wanlix 中，寄存器组是在任务切换的时候被保存在栈指针当前的位置的，在 Mindows 中我把它放到了 TCB 里面，这样做的好处是寄存器组被固定在 TCB 中固定的位置，使用的时候可以很方便的通过 TCB 找到，而且，在 Mindows 中也不可能像 Wanlix 那样把寄存器组放在任务切换时的栈指针处，因为 Mindows 的任务切换是由中断引起的，中断产生的时机是不可预知的，如果产生中断时 SP 栈指针没有指向当时栈的位置，那么将寄存器组存储在 SP 栈指针指向的内存则会破坏内存中的数据，可能会导致系统崩溃。

在中断发生时，芯片内核会从 USR 模式切换到 IRQ 模式（这里使用的 tick 中断是 IRQ 模式的），中断返回的下条指令地址被保存到了 IRQ 模式下的 LR 寄存器中，这个值相当于是 PC 值，而 USR 模式下的 LR 寄存器值没有变，它保存的是 USR 模式下当前函数返回上级父函数的 PC 值。因此，Mindows 在任务上下文切换时要比 Wanlix 多备份、恢复一个 PC 寄存器，在 Mindows 中，寄存器组就由 CPSR、R0~R15 组成了：

```
typedef struct stackreg
{
    U32 uiCpsr;
    U32 uiR0;
    U32 uiR1;
    U32 uiR2;
    U32 uiR3;
    U32 uiR4;
    U32 uiR5;
    U32 uiR6;
    U32 uiR7;
    U32 uiR8;
    U32 uiR9;
    U32 uiR10;
    U32 uiR11;
    U32 uiR12;
    U32 uiR13;
    U32 uiR14;
    U32 uiR15;
}STACKREG;
```

我们再来看看 TCB 中的 M_TCBQUE 结构体，

```
typedef struct m_tcbque             /* TCB 队列结构 */
{
    M_CHAIN strQueHead;             /* 连接队列的链表 */
    struct m_tcb* pstrTcb;          /* TCB 指针 */
}M_TCBQUE;
```

M_TCBQUE 队列结构相比 M_CHAIN 链表结构来说，只是多了一个 M_TCB 型的指针 pstrTcb。我们从 ready 表中获取最高优先级任务时，先获取到最高优先级，再通过最高优先级获取到链表根节点，再通过根节点获取到挂在上面的任务的子节点，然后应该就是再通过子节点获取到任务的 TCB。pstrTcb 指针在任务初始化时就被初始化为任务的 TCB 指针，因此在获取到子节点 strQueHead 的地址后，我们就可以通过它下面的 pstrTcb 获取到 TCB 的地址，进而可以获取整个 TCB 的信息。

本节中新增的重要内容已经介绍的差不多了，下面我们来看看使用 tick 中断的任务上下文切换过程。

在本节中，tick 中断被配置为使用 Timer1 产生 IRQ 中断。产生 tick 中断时，硬件会自动将 PC 指针指向 IRQ 中断向量表，IRQ 中断向量里存放的是一条跳转指令，会跳转到 IRQ 中断服务函数，这个中断服务函数需要由我们来编写，它的功能就是备份、恢复寄存器组。在 Mindows 里我们使用 MDS_TickContextSwitch 函数来实现这个功能，在 start.s 文件里需要把中断向量表的 IRQ 中断服务函数修改为 MDS_TickContextSwitch。

由于 C 函数会自动生成入栈出栈的指令，任务切换时调用 C 函数会破坏寄存器组中的数据，因此，MDS_TickContextSwitch 函数需要使用汇编语言来写。备份、恢复寄存器组的原理与 Wanlix 中讲述是一样的，由于是在 IRQ 中断中执行的，实现的细节上还是有区别的，我们来看代码：

```
00013       .func MDS_TickContextSwitch
00014   MDS_TickContextSwitch:
00015
00016       @保存接口寄存器
00017       STMDB   R13!, {R0 - R3, R12, R14}
00018
00019       @调用 C 语言 TICK 中断处理函数
00020       LDR     R0, =MDS_TickIsr
00021       MOV     R14, PC
00022       BX      R0
00023
00024       @保存当前任务的堆栈信息
00025       LDR     R0, =gpstrCurTaskSpAddr
00026       LDR     R14, [R0]
00027       MRS     R0, SPSR
00028       STMIA   R14!, {R0}
00029       LDMIA   R13!, {R0 - R3, R12}
00030       STMIA   R14, {R0 - R14}^
00031       ADD     R14, R14, #0x3C
00032       LDMIA   R13!, {R0}
00033       STMIA   R14, {R0}
00034
00035       @任务调度完毕，恢复将要运行任务现场
00036       LDR     R0, =gpstrNextTaskSpAddr
00037       LDR     R14, [R0]
00038       LDMIA   R14, {R0}
00039       MSR     SPSR, R0
00040       LDMIB   R14, {R0 - R14}^
00041       NOP
00042       ADD     R14, R14, #0x40
00043       LDMIA   R14, {R14}
00044       SUBS    PC, R14, #4
00045
00046       .endfunc
```

00017 行，保存切换前任务的寄存器。本函数在第 22 行会调用 MDS_TickIsr 函数，在 2.3 节中我们已经介绍了 AAPCS 参数传递的规则，MDS_TickIsr 函数可能会破坏接口寄存器，因此在本函数中需要保存 R0~R3、R12 寄存器。其余的寄存器，本函数在备份它们之前并没有破坏它们的数据，因此不需要备份。这段代码是在 IRQ 模式下执行的，此时的 SP 是 SP_{IRQ}，数据被保存到 IRQ 的栈中。

00020~00022 行，执行 MDS_TickIsr 函数。MDS_TickIsr 函数是使用 C 语言写的函数，这个函数里才是真正的 IRQ 中断处理函数，如果不使用操作系统，这个函数是应该直接挂到 IRQ 中断向量表上的。MDS_TickIsr 函数会判断产生 IRQ 的中断源，根据 IRQ 不同的中断源走不同的程序分支，其中包括产生 tick 中断的 Timer1 中断。MDS_TickIsr 函数最终调

用了 MDS_TaskSched 函数，做任务切换前的准备工作，包括获取将要运行的最高优先级任务，将切换前任务和切换后任务的寄存器组所在的地址存入全局变量 gpstrCurTaskSpAddr 和 gpstrNextTaskSpAddr 中等操作。

00025 行，将全局变量 gpstrCurTaskSpAddr 的地址放入 R0，gpstrCurTaskSpAddr 的内容是切换前任务的寄存器组存放在栈中的地址。

00026 行，将切换前任务的寄存器组存放在栈中的地址赋给 LR 寄存器，注意，这段程序是在 IRQ 中断中执行的，这个 LR 是 LR_{IRQ}。

00027 行，将 $SPSR_{IRQ}$ 也就是 $CPSR_{USR}$ 的值存入 R0。

00028 行，将 R0 的内容存入 LR_{IRQ} 指向的地址，也就是将 $CPSR_{USR}$ 存入切换前任务的寄存器组所在栈中的第一个地址，也就是将切换前任务的 $CPSR_{USR}$ 存入到切换前任务 TCB 中的寄存器组中的 CPSR 的位置，这条指令就是备份寄存器组中的 CPSR。

00029 行，从 IRQ 堆栈中恢复 R0~R3 和 R12 寄存器，执行完这条指令，USR 模式下的 R0~R14 都已经恢复为中断发生那一时刻的值了。

00030 行，将 USR 模式下的 R0~R14 备份到栈中的寄存器组中对应的位置。这条指令的第一个参数 LR 是 LR_{IRQ}，指向的是切换前任务的栈中寄存器组的 R0 所在位置，第二个参数里的 LR 是 LR_{USR}，因为这条指令里有“^”符号，这个我们在 2.2 节中介绍过。

00031 行，将 LR_{IRQ} 更新到栈中的寄存器组的 PC 所在位置。

00032 行，从 IRQ 栈中取出中断发生时的 LR_{IRQ}。在 17 行时向 IRQ 栈存入了 6 个寄存器，在 29 行取出了 5 个，剩下的最后一个 LR_{IRQ} 在这里取出，它的值就是中断返回时的 PC 值。

00033 行，将 LR_{IRQ} 存入切换前任务的栈中寄存器组的 PC 位置，也就是将中断返回的 PC 值存入栈中寄存器组的 PC 位置。

自此，备份寄存器组的工作已经完成。

00036 行，将全局变量 gpstrNextTaskSpAddr 的地址放入 R0，gpstrNextTaskSpAddr 的内容是切换后任务的寄存器组存放在栈中的地址。

00037 行，将切换后任务的寄存器组存放在栈中的地址赋给 LR_{IRQ} 寄存器。

00038 行，从切换后任务的栈中寄存器组取出 CPSR 的值，存入 R0。

00039 行，将 CPSR 的值赋给 $SPSR_{IRQ}$，这就是恢复 CPSR 寄存器的过程，但真正恢复 CPSR 的操作还需要从 IRQ 模式返回到 USR 模式时才能执行，那时候硬件会自动将 $SPSR_{IRQ}$ 中数值恢复到 CPSR 中。

00040 行，从栈中寄存器组中恢复 USR 模式下的 R0~R14 寄存器，此时，USR 模式下的 R0~R14 已经全部恢复为该任务上次切出去时候的值了。

00041 行，NOP，指令空闲一个周期什么也不做。别小看这个 NOP 指令，当初就因为没有这条指令 Mindows 程序老是跑飞，卡住了我好几天，最后才定位到这里，但又找不到原因。观察 41 行和 42 行，这两行指令里都使用了 LR 寄存器，而且是 LR_{USR} 和 LR_{IRQ} 连在一起使用的，估计可能是芯片内部指令总线上起了冲突才导致程序跑飞的，最后加了这条 NOP 指令程序才正常，但一直不知道为什么，没有查到相关的资料，哪位知道的话请到论坛上反馈一下，多谢！求真相！

00042 行，将 LR_{IRQ} 更新到切换后任务的栈中的寄存器组的 PC 所在位置。

00043 行，将返回的 PC 值存入 LR_{IRQ}。

00044 行，跳转到 USR 模式下中断前的下条指令继续执行，同时将 $SPSR_{IRQ}$ 中的数值恢复到 CPSR 中。IRQ 中断发生时，会将刚执行完的指令的地址＋8 存入到 LR_{IRQ} 中，当中

断返回时，需要执行的是＋4 地址的指令，因此，从 IRQ 返回时需要使用 SUBS 指令将再 LR_{IRQ}－4 存入到 PC 中，这样才能正好跳转到中断前的下条指令继续执行。这个过程在参考资料 2 中有描述。

下面我们来跟踪一个 tick 中断产生的过程，用以了解 Mindows 任务的调度流程，见图 37：

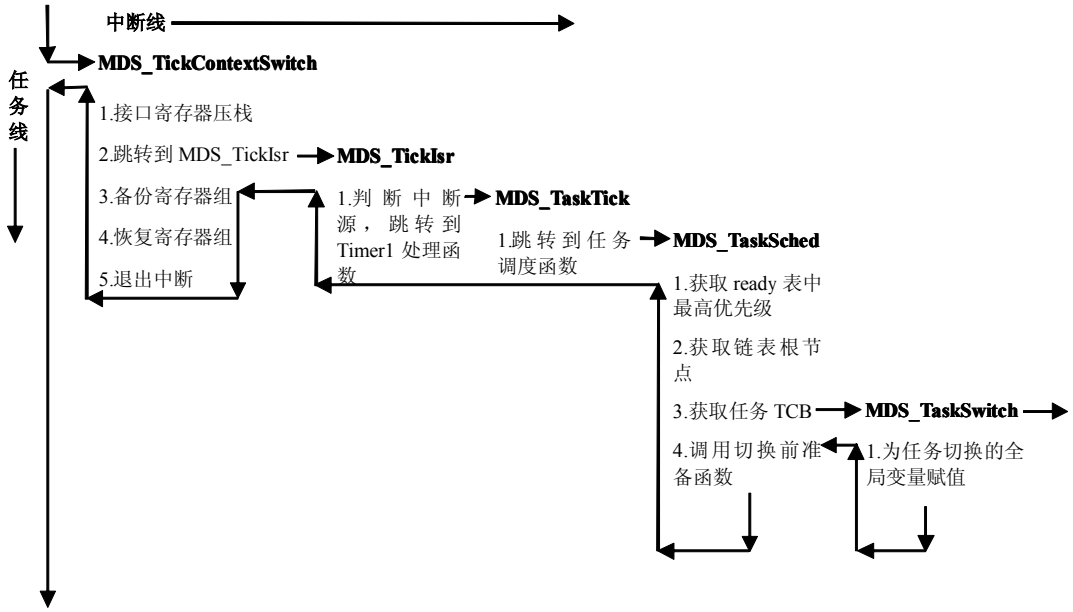


图 37　Mindows 任务调度流程

MDS_TickIsr、MDS_TaskTick 函数比较简单，MDS_TaskSwitch 函数与 Wanlix 中的这个函数没有本质的区别，这 3 个函数就不介绍了。我们来看看任务调度函数 MDS_TaskSched：

```
00209  void MDS_TaskSched(void)
00210  {
00211      M_TCB* pstrTcb;
00212      M_TCBQUE* pstrTaskQue;
00213      U8 ucTaskPrio;
00214
00215      /* 获取 ready 表中优先级最高的任务的 TCB */
00216      ucTaskPrio = MDS_TaskHighestPrioGet(&gstrReadyTab.strFlag);
00217      pstrTaskQue = (M_TCBQUE*)MDS_ChainEmpInq(&gstrReadyTab.astrChain[ucTaskPrio]);
00218      pstrTcb = pstrTaskQue->pstrTcb;
00219
00220      /* 准备任务切换 */
00221      MDS_TaskSwitch(pstrTcb);
00222  }
```

00216 行，从 ready 表标志中获取最高的优先级。

00217 行，获取最高优先级对应的链表根节点，并将根节点的 M_CHAIN 型指针强制转换成 M_TCBQUE 型指针。

00218 行，通过 M_TCBQUE 指针类型获取任务 TCB 的指针。

00221 行，调用 MDS_TaskSwitch 函数，做寄存器组备份、恢复前的准备工作。

由于增加了 ready 表，在创建任务时需要对相关的变量进行初始化，MDS_TaskCreate

函数代码如下：

```
00015  M_TCB* MDS_TaskCreate(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize,
00016                        U8 ucTaskPrio)
00017  {
00018      M_TCB* pstrTcb;
00019
00020      /* 对创建任务所使用函数的指针合法性进行检查 */
00021      if(NULL == vfFuncPointer)
00022      {
00023          /* 指针为空，返回失败 */
00024          return (M_TCB*)NULL;
00025      }
00026
00027      /* 对任务堆栈合法性进行检查 */
00028      if((NULL == pucTaskStack) || (0 == uiStackSize))
00029      {
00030          /* 堆栈不合法，返回失败 */
00031          return (M_TCB*)NULL;
00032      }
00033
00034      /* 对是否重复创建 root 任务进行检查 */
00035      if(MDS_RootTask == vfFuncPointer)
00036      {
00037          /* root 任务已经创建过，返回失败 */
00038          if(NULL != gpstrRootTaskTcb)
00039          {
00040              return (M_TCB*)NULL;
00041          }
00042      }
00043
00044      /* 初始化 TCB */
00045      pstrTcb = MDS_TaskTcbInit(vfFuncPointer, pucTaskStack, uiStackSize, ucTaskPrio);
00046
00047      return pstrTcb;
00048  }
```

00015 行，函数返回值是新创建任务的 TCB 指针，若为 NULL，代表创建任务失败，其它值则为创建任务成功，为新任务的 TCB 指针。入口参数 ucTaskPrio 是新创建任务的优先级，其它入口参数与以前一致，没有变化。

00021~00025 行，入口参数判断，若创建任务的函数指针为 NULL，返回失败。

00028~00032 行，入口参数判断，若创建任务的堆栈不合法，返回失败。

00035~00042 行，若重复创建根任务，则返回失败。根任务只能创建一次，在操作系统初始函数 MDS_SystemVarInit 里会将根任务的 TCB 指针 gpstrRootTaskTcb 初始化为 NULL，当根任务创建成功后 gpstrRootTaskTcb 就被更改为根任务的 TCB 值，即便根任务运行结束后 gpstrRootTaskTcb 中保持的 TCB 值仍然不变，在创建任务时可根据 gpstrRootTaskTcb 来判断根任务是否被重复创建。

00045 行，初始化任务的 TCB。

00047 行，任务创建成功，返回任务的 TCB 指针。

建立任务时的初始化操作主要集中在 MDS_TaskTcbInit 函数里面，由于寄存器组被放到了 TCB 里面，因此 MDS_TaskStackInit 函数也由 MDS_TaskTcbInit 函数调用，来看一下 MDS_TaskTcbInit 函数的代码：

```
00059  M_TCB* MDS_TaskTcbInit(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize,
00060                         U8 ucTaskPrio)
00061  {
00062      M_TCB* pstrTcb;
00063      M_CHAIN* pstrChain;
00064      M_CHAIN* pstrNode;
00065      M_PRIOFLAG* pstrPrioFlag;
00066      U8* pucStackBy4;
00067
00068      /* 堆栈满地址，需要 4 字节对齐 */
00069      pucStackBy4 = (U8*)(((U32)pucTaskStack + uiStackSize) & 0xFFFFFFFC);
00070
00071      /* TCB 结构存放的地址，需要 4 字节对齐 */
00072      pstrTcb = (M_TCB*)(((U32)pucStackBy4 - sizeof(M_TCB)) & 0xFFFFFFFC);
00073
00074      /* 初始化任务堆栈 */
00075      MDS_TaskStackInit(pstrTcb, vfFuncPointer);
00076
00077      /* 初始化指向 TCB 的指针 */
00078      pstrTcb->strTcbQue.pstrTcb = pstrTcb;
00079
00080      /* 初始化任务优先级 */
00081      pstrTcb->ucTaskPrio = ucTaskPrio;
00082
00083      pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00084      pstrNode = &pstrTcb->strTcbQue.strQueHead;
00085      pstrPrioFlag = &gstrReadyTab.strFlag;
00086
00087      /* 锁中断，防止其它任务影响 */
00088      (void)MDS_IntLock();
00089
00090      /* 将该任务添加到 ready 表中 */
00091      MDS_TaskAddToSchedTab(pstrChain, pstrNode, pstrPrioFlag, ucTaskPrio);
00092
00093      /* 挂入链表后解锁中断，允许任务调度 */
00094      (void)MDS_IntUnlock();
00095
00096      return pstrTcb;
00097  }
```

00059 行，函数返回值是新创建任务的 TCB 指针，若为 NULL，代表创建任务失败，其它值则为创建任务成功，为新任务的 TCB 指针。

00075 行，初始化任务栈。

00078 行，将任务 TCB 指针赋给 TCB 队列中的指针变量。

00081 行，将任务的优先级保存到 TCB 中。

00083 行，根据任务优先级，获取 ready 中同等的优先级链表根节点指针。

00084 行，获取 TCB 中可加入 ready 链表的链表结构指针。

00085 行，获取 ready 表的标志结构指针。

00088 行，锁 Timer1 中断，防止执行下面的代码时发生任务调度。在 91 行会对 ready 表进行操作，修改它的链表和标志，这个过程是要分几个步骤进行的，ready 表及其上面挂接的各个任务节点对所有任务都是同时可见的，如果在这过程中切换到了其它任务，而其它任务也来对 ready 表进行操作，那么 ready 表就可能会因为多个任务同时修改它的数据而导致数据异常。因此，我们要防止这种情况发生，必须保证每次只有一个任务对 ready 操作，当这个任务对 ready 表操作完成之后才能允许其它任务再次操作 ready 表。为了实现这个功能，我们可以使用 MDS_IntLock 函数将 tick 中断锁住，这样就不能产生 tick 中断了，因此

也就不会发生任务调度了，因此也就保证了只有一个任务可以对 ready 进行操作，保证了 ready 表操作的串行性。在 ready 表操作完之后，需要调用 MDS_IntUnlock 函数打开 tick 中断，这样操作系统又可以继续进行任务调度了。注意，锁 tick 中断的过程中操作系统丧失了任务调度功能，因此锁 tick 中断的时间一定要尽可能的短。不但是 tick 中断，不但是在操作系统状态下，任何情况下，我们都需要将任何锁中断的时间做的尽可能的短。

00091 行，将新建立的任务挂接到 ready 表对应的优先级链表中，新建立的任务处于 ready 态，准备运行。

00094 行，解锁中断，恢复任务调度功能。

00096 行，任务创建成功，返回新建任务的 TCB 指针。

前面说过进入 IRQ 时，存入 LR_{IRQ} 的是刚执行完的指令地址＋8，在退出中断时需要使用 SUBS 指令将 LR_{IRQ}－4 存入 PC 寄存器，但在 USR 模式下函数调用时是没有这种＋8 和－4 操作的，而 Mindows 操作系统从非操作系统状态转换为操作系统状态正是在 USR 模式下依靠函数调用实现的，开始运行第一个任务 MDS_RootTask，但其它的任务第一次调用时则是依靠 IRQ 中断开始调度的，因此，在初始化寄存器组中的 PC 值时要分别对待，来看栈初始化函数 MDS_TaskStackInit 的代码：

```
00011  void MDS_TaskStackInit(M_TCB* pstrTcb, VFUNC vfFuncPointer)
00012  {
00013      STACKREG* pstrRegSp;
00014
00015      pstrRegSp = &pstrTcb->strStackReg;          /* 寄存器组地址 */
00016
00017      /* 对 TCB 中的寄存器组初始化 */
00018      pstrRegSp->uiCpsr = MODE_USR;               /* CPSR */
00019      pstrRegSp->uiR0 = 0;                        /* R0 */
00020      pstrRegSp->uiR1 = 0;                        /* R1 */
00021      pstrRegSp->uiR2 = 0;                        /* R2 */
00022      pstrRegSp->uiR3 = 0;                        /* R3 */
00023      pstrRegSp->uiR4 = 0;                        /* R4 */
00024      pstrRegSp->uiR5 = 0;                        /* R5 */
00025      pstrRegSp->uiR6 = 0;                        /* R6 */
00026      pstrRegSp->uiR7 = 0;                        /* R7 */
00027      pstrRegSp->uiR8 = 0;                        /* R8 */
00028      pstrRegSp->uiR9 = 0;                        /* R9 */
00029      pstrRegSp->uiR10 = 0;                       /* R10 */
00030      pstrRegSp->uiR11 = 0;                       /* R11 */
00031      pstrRegSp->uiR12 = 0;                       /* R12 */
00032      pstrRegSp->uiR13 = (U32)pstrTcb;            /* R13 */
00033      pstrRegSp->uiR14 = 0;                       /* R14 */
00034
00035      /* 非 root 任务首次运行可能会在 IRQ 中断里跳转执行，函数入口地址需要多加 4 字节 */
00036      if(MDS_RootTask != vfFuncPointer)
00037      {
00038          pstrRegSp->uiR15 = (U32)vfFuncPointer + 4;  /* R15 */
00039      }
00040      else    /* root 任务在 USE 模式下直接执行，可以直接跳转到函数地址 */
00041      {
00042          pstrRegSp->uiR15 = (U32)vfFuncPointer;      /* R15 */
00043      }
00044  }
```

00011~00033 行，初始化寄存器组中的 CPSR、R0~R14，与以前一致，没有变化。

00036~00039 行，判断创建的若不是 root 任务，则将创建任务所用函数的指针＋4 存入

寄存器组中 PC 的位置。若创建的不是 root 任务，需要由 IRQ 中断开始调度，在 IRQ 中断返回时会使用 SUBS 指令将返回的 PC 地址－4，因此创建非 root 任务需要将 PC 多＋4。

00042 行，创建的任务是 root 任务，将创建任务所用函数的指针直接存入寄存器组中 PC 的位置。因为 root 任务是在 USR 模式下采用函数调用的方式开始运行的，不需要多＋4。

上面介绍了本节中关键部分的代码，其它代码请读者自行参考源代码，这里不再介绍了。接下来，我们使用测试函数来看看本节的成果。

本节只引入了 ready 表，没有其它可以控制任务调度的方法，一旦最高优先级任务开始运行就无法切换到其它任务了。

我们使用 3 个测试函数 TEST_TestTask1、TEST_TestTask2 和 TEST_TestTask3，每个函数都是循环执行“打印字符串，延迟时间”的操作。

```
00015  void TEST_TestTask1(void)
00016  {
00017      while(1)
00018      {
00019          DEV_PutString((U8*)"\r\nTask1 is running!");
00020
00021          DEV_DelayMs(1000);              /* 延迟 1s */
00022      }
00023  }

00030  void TEST_TestTask2(void)
00031  {
00032      while(1)
00033      {
00034          DEV_PutString((U8*)"\r\nTask2 is running!");
00035
00036          DEV_DelayMs(2000);              /* 延迟 2s */
00037      }
00038  }

00045  void TEST_TestTask3(void)
00046  {
00047      while(1)
00048      {
00049          DEV_PutString((U8*)"\r\nTask3 is running!");
00050
00051          DEV_DelayMs(3000);              /* 延迟 3s */
00052      }
00053  }
```

这 3 个函数都由 MDS_RootTask 任务创建，创建后都延迟 1 秒。

```
00013  void MDS_RootTask(void)
00014  {
00015      /* 初始化软件 */
00016      DEV_SoftwareInit();
00017
00018      /* 初始化硬件 */
00019      DEV_HardwareInit();
00020
00021      /* 创建任务 */
00022      (void)MDS_TaskCreate((VFUNC)TEST_TestTask1, gaucTask1Stack, TASKSTACK, 3);
00023
```

```
00024       DEV_DelayMs(1000);
00025
00026       (void)MDS_TaskCreate((VFUNC)TEST_TestTask2, gaucTask2Stack, TASKSTACK, 2);
00027
00028       DEV_DelayMs(1000);
00029
00030       (void)MDS_TaskCreate((VFUNC)TEST_TestTask3, gaucTask3Stack, TASKSTACK, 0);
00031
00032       DEV_DelayMs(1000);
00033   }
```

我们将 MDS_RootTask 任务的优先级设为 1，TEST_TestTask1 任务的优先级设定为 3，TEST_TestTask2 任务的优先级设定为 2，TEST_TestTask3 任务的优先级设定为 0，我们按照本节的任务调度方式，从系统上电运行开始，推算一下任务的切换过程，如图 38：

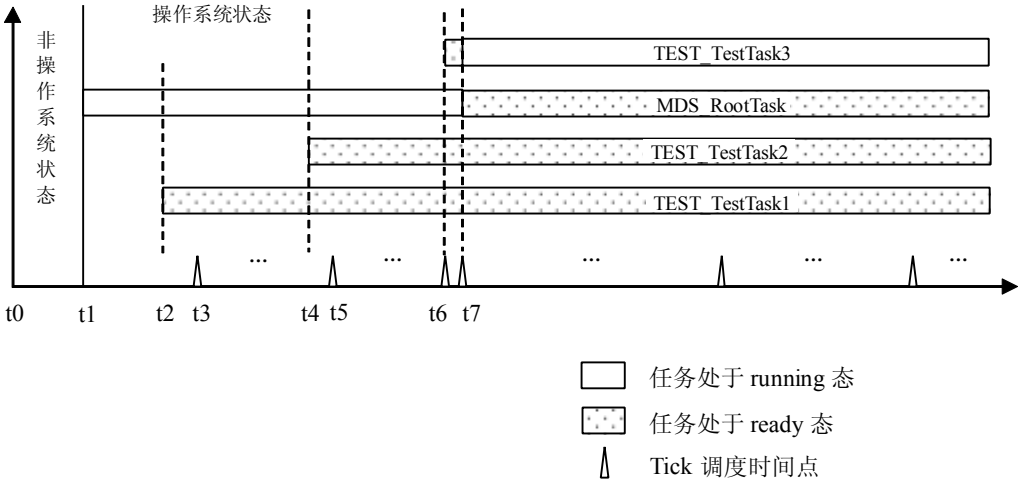


图 38　4.2 节测试任务执行过程

t0 时刻，系统开始运行。

t1 时刻，从非操作系统状态切换到操作系统状态，开始运行 MDS_RootTask 任务，这时候 MDS_RootTask 任务处于 running 态。在 MDS_RootTask 任务里初始化了 tick 中断，在 tick 中断里开始任务调度，此时，ready 表中只有 MDS_RootTask 一个任务。

t2 时刻，创建了 TEST_TestTask1 任务，此时 ready 表中有 MDS_RootTask 和 TEST_TestTask1 共 2 个任务，现在还是在运行 MDS_RootTask 任务。MDS_RootTask 任务处于 running 态，TEST_TestTask1 任务处于 ready 态。

t3 时刻，tick 中断到来，调度任务，由于 MDS_RootTask 任务比 TEST_TestTask1 任务的优先级高，因此调度的结果还是执行 MDS_RootTask 任务。MDS_RootTask 任务处于 running 态，TEST_TestTask1 任务处于 ready 态。

t3~t4 之间不断产生 tick 中断，调度任务，由于 MDS_RootTask 任务比 TEST_TestTask1 任务的优先级高，因此调度的结果还是执行 MDS_RootTask 任务。MDS_RootTask 任务处于 running 态，TEST_TestTask1 任务处于 ready 态。

t4 时刻，创建了 TEST_TestTask2 任务，此时 ready 表中有 MDS_RootTask、TEST_TestTask1 和 TEST_TestTask2 共 3 个任务，现在还是在运行 MDS_RootTask 任务。MDS_RootTask 任务处于 running 态，TEST_TestTask1 和 TEST_TestTask2 任务处于 ready 态。

t5 时刻，tick 中断到来，调度任务，由于 MDS_RootTask 任务优先级最高，因此调度的结果还是执行 MDS_RootTask 任务。MDS_RootTask 任务处于 running 态，TEST_TestTask1 和 TEST_TestTask2 任务处于 ready 态。

t5~t6 之间不断产生 tick 中断，调度任务，由于 MDS_RootTask 任务优先级最高，因此调度的结果还是执行 MDS_RootTask 任务。MDS_RootTask 任务处于 running 态，TEST_TestTask1 和 TEST_TestTask2 任务处于 ready 态。

t6 时刻，创建了 TEST_TestTask3 任务，此时 ready 表中有 MDS_RootTask、TEST_TestTask1、TEST_TestTask2 和 TEST_TestTask3 共 4 个任务，现在还是在运行 MDS_RootTask 任务。MDS_RootTask 任务处于 running 态，TEST_TestTask1 和 TEST_TestTask2 和 TEST_TestTask3 任务处于 ready 态。

t7 时刻，tick 中断到来，调度任务，由于 TEST_TestTask3 任务优先级最高，因此发生任务切换，TEST_TestTask3 任务从 ready 态变为 running 态，而 MDS_RootTask 任务则从 running 态变为 ready 态，调度的结果变为执行 TEST_TestTask3 任务了。TEST_TestTask3 任务处于 running 态，MDS_RootTask、TEST_TestTask1 和 TEST_TestTask2 任务处于 ready 态。

t7 之后不断产生 tick 中断，调度任务，由于 TEST_TestTask3 任务优先级最高，因此调度的结果还是执行 TEST_TestTask3 任务。TEST_TestTask3 任务处于 running 态，MDS_RootTask、TEST_TestTask1 和 TEST_TestTask2 任务处于 ready 态。

TEST_TestTask3 任务每隔 3 秒打印一次，因此，我们最终从串口打印看到的只能是 TEST_TestTask3 任务每隔 3 秒一次的打印，而看不到 TEST_TestTask1 和 TEST_TestTask2 任务的打印。

下面是本节输出的打印截图：

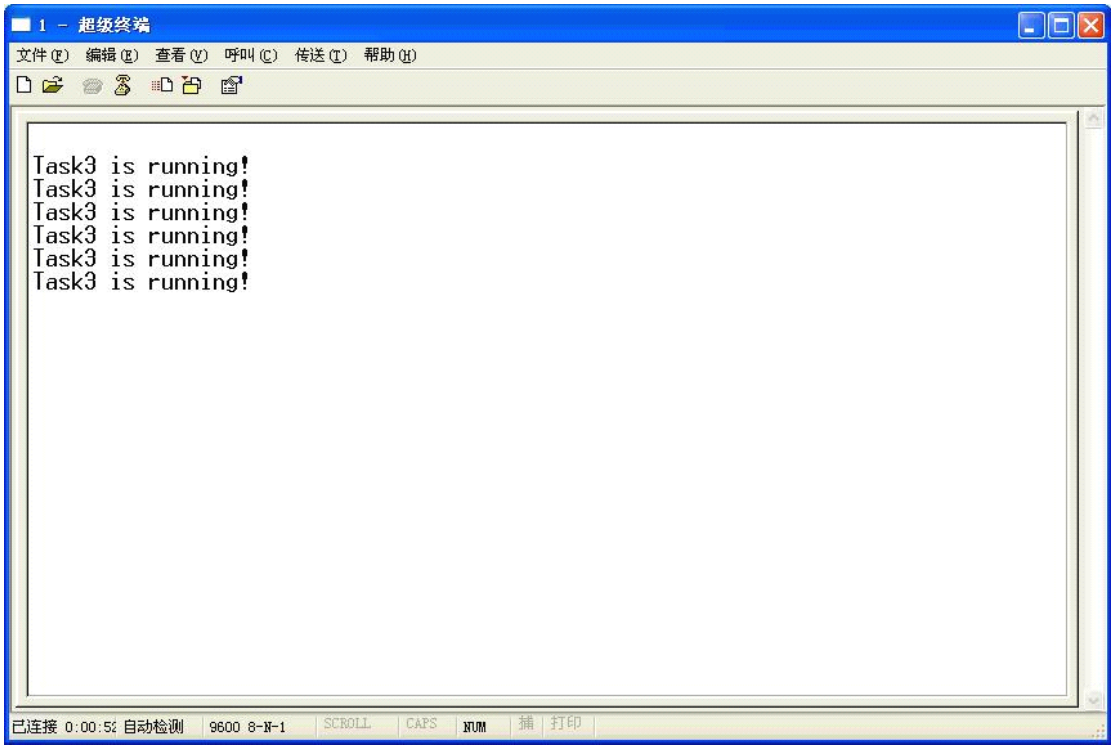


图 39 tick 中断调度任务的结果

大家可以到 http://blog.sina.com.cn/ifreecoding 网站下载视频，观看本节的动态打印输出。从视频中可以看到 TEST_TestTask3 任务每隔 3 秒打印一次，与我们分析的情况一致。

从图 38 中可以看到，拥有最高优先级的 TEST_TestTask3 任务在 t6 时刻就已经创建了，但还需要等到 t7 时刻 tick 中断到来时才能够执行，实时性还是差了一些。实时操作系统的

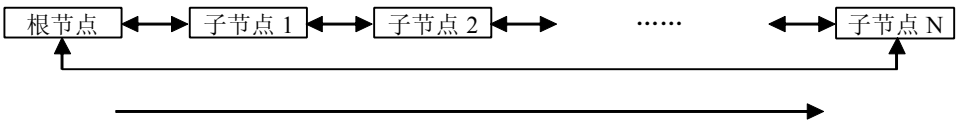
调度周期并非完全是由 tick 周期决定，在下节我们将了解实时操作系统的另一种调度方式——实时事件触发的随机调度，这种调度方式对提高操作系统的实时性也是有帮助的。

第 3 节 实时事件触发的实时抢占调度

在上节中我们成功的实现了任务的 ready 状态，并用 tick 中断实现了实时调度，但由于只有 ready 这一种状态，使得我们的例子只能不断的运行最高优先级任务。在这一节我们引入任务的另一个状态——delay 状态，当前运行的任务可以通过调用 MDS_TaskDelay 函数进入 delay 状态，将 CPU 控制权交给其它任务，经过一段时间后，它可以再恢复为 ready 状态，重新参与任务调度。

本小节将使用 ready 状态和 delay 状态在多个任务之间按任务优先级实现任务交替运行，输出多个任务的打印结果。

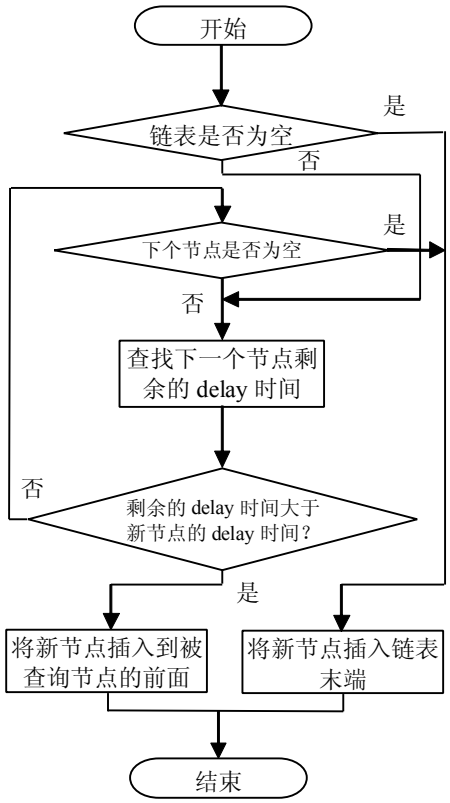
ready 表关联了处于 ready 状态的任务，处于 delay 状态的任务也需要使用一个 delay 表来关联，ready 表需要使用任务优先级这个属性来调度其中的各个任务，而 delay 表则是以需要 delay 的时间这个属性来调度其中任务的，按照需要 delay 时间的长短，从短到长，将处于 delay 状态的各个任务节点挂接到 delay 表的根节点上。



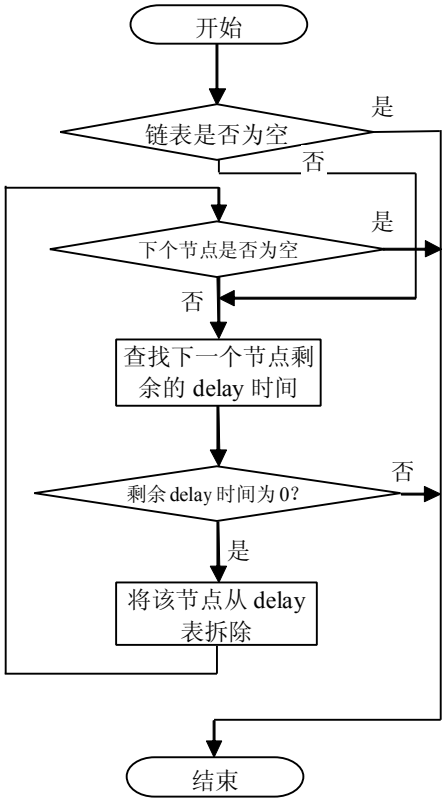
链表节点按 delay 时间从短到长顺序连接

图 40 delay 表结构

delay 表相对 ready 表来说要简单很多，它只有一个根节点。delay 表初始化时，根节点被初始化为空。向 delay 表添加节点时，先从根节点找到第一个子节点，若该子节点剩余的 delay 时间小于等于需要新添加节点的 delay 时间，则继续查找下一个节点，直到找到节点的剩余 delay 时间大于新添加节点的 delay 时间时，将新节点添加到这个节点的前面，若链表中所有节点剩余的 delay 时间都小于等于新添加节点的 delay 时间，则将新添加的节点挂到 delay 表的最后。由于 delay 表是按照 delay 时间从短到长的顺序排列的，当从 delay 表拆除节点时，只需要从 delay 表的第一个节点开始向后找，找到最后一个剩余的 delay 时间不为 0 的节点为止，将这些剩余的 delay 时间为 0 的节点全部从 delay 表中拆除，挂入到 ready 表即可。



向 delay 表添加节点的操作流程



从 delay 表拆除节点的操作流程

图 41　delay 表操作流程图

需要永久 delay 的任务不挂入 delay 表，因为它与时间不相关，不需要参与 tick 调度。

任务 delay 的时间是以 tick 为单位的，每次 tick 中断产生时，在函数 MDS_TaskTick 里会对 guiTick 全局变量做加 1 操作，操作系统就是以这个 guiTick 变量作为操作系统的系统时钟的。

任务调用 MDS_TaskDelay(uiDelayTick)函数时，并不会真正延迟 uiDelayTick 个 ticks 的时间，这是因为任务是在 2 个 ticks 之间调用 MDS_TaskDelay 函数的，调用 MDS_TaskDelay 函数的时刻到下次 tick 中断的时刻之间的时间是不够 1 个 tick 时间的，因此，任务 delay 的全部时间只有(uiDelayTick－1)~uiDelayTick 个 ticks 时间。如果你希望至少延迟 N 个 ticks 的话，那么就需要将 MDS_TaskDelay 的参数设置为(N＋1)，如果 delay 的时间为 0，那么该任务并不会进入 delay 状态，操作系统只是将 ready 表中的任务重新调度了一次而已。

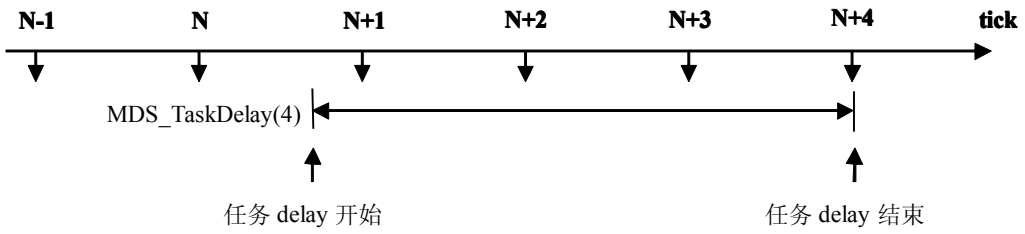


图 42　任务 delay 时间

在图 42 中，任务调用 MDS_TaskDelay 函数时从 running 态转变为了 delay 态，那么操作系统就需要从 ready 表中找到一个最高优先级的任务将它转换成 running 态，但任务调度的过程是在 tick 中断中发生的，而此时并没有发生 tick 中断，因此，我们就需要增加一种新的任务调度触发方式，增加实时事件触发的任务调度。

在 Mindows 中，我们使用软中断来实现这个功能，在 tick 中断之外出现的实时事件需要触发任务调度时，它会触发软中断，继而运行软中断服务函数，在软中断服务函数里做一个任务调度，软中断服务函数里面所做的调度工作与 tick 中断服务函数里所做的调度工作非常相似。

处于 delay 状态的任务除了可以通过将需要 delay 的时间耗尽，从 delay 状态返回到 ready 状态，还可以通过其它任务调用 MDS_TaskWake 函数来唤醒该任务，使处于 delay 状态的任务立刻变为 ready 态，参与到 ready 表的任务调度中。

在前面章节的测试函数中，我们所使用的 DEV_DelayMs 函数会连续运行几秒钟时间，在这几秒时间内任务一直处于 running 态，一直占有 CPU，用来模拟当前任务正在运行的业务。但在实际的项目中，这种情况是不可以存在的，每个任务连续运行的时间不会很长，每运行若干 ms 甚至 us 的时间就会让出 CPU 资源，由其它任务继续运行。

在一段时间内，与整个系统功能相关的指令运行所花费的时间，与这段时间的比值叫做 CPU 占有率，从这个定义可以看出，CPU 占有率越高 CPU 剩下的可使用的处理能力越少。实时操作系统实时抢占的特点是具有随机性的，我们不知道什么时候就会有一个更高优先级的任务抢占了当前正在执行的任务，如果此时 CPU 占有率过高，那么突发的一些较高优先级任务就会使 CPU 占有率接近 100%，而此时这些较高优先级任务中的一些任务则可能会由于得不到 CPU 资源而无法运行，这样系统就会丧失一些重要功能，从而影响整个系统的功能。当我们设计产品软硬件系统时，需要根据产品的功能、性能选择合适的处理器，在空闲状态时，要保证 CPU 占有率处于较低的水平，留有足够的 CPU 余量以备满业务负载或突发事件使用。

在系统运行过程中，很可能会出现所有的任务同时处于 delay 状态的情况，没有 ready 态的任务，那么操作系统也就无法从 ready 表中找出一个可以转换为 running 态的任务去执行，那么，这时候 CPU 应该执行什么代码呢？Mindows 操作系统在初始化时，除了会创建 root 根任务，还会创建一个 idle 空闲任务，当所有其它任务都不处于 ready 态时，操作系统就运行 idle 任务。CPU 占有率也可以认为是一段时间内非空闲任务运行的时间与这段时间的比值，运行 idle 任务时 CPU 虽然也是在执行指令，但这些指令是不算在 CPU 占有率中的，因此在 idle 任务中不要放入具有业务功能的代码，并且需要保证 idle 任务具有最低的优先级，使它不能影响其它任务的执行。在 Mindows 中，将最低优先级保留给 idle 任务，其它任务不能使用该优先级。root 任务是系统最先运行的任务，它应该先于其它任务运行完毕，因此 Mindows 将最高优先级保留给 root 任务，其它任务不能使用该优先级。

idle 任务不允许处于 delay 状态，如果 idle 任务也被 delay 了，那么操作系统就真的不知道该做什么了。Mindows 对 MDS_TaskDelay 函数做了判断，如果用户在 idle 任务里误调用了 MDS_TaskDelay 则什么情况也不会发生，相当于是调用了空函数，这样可以防止操作系统崩溃。idle 任务永远处于 ready 状态。

Mindows 内核中有一些函数不可重入，在使用这些函数前需要先锁中断，使用之后再解锁中断，这样就可以防止多个任务并行运行这些函数发生重入。上节中的锁中断函 数

MDS_IntLock 和解锁中断函数 MDS_IntUnlock 只针对 tick 中断做了处理，只能阻止 tick 中断产生的重入现象，却不能阻止在其它中断中调用这些函数产生的函数重入。因此从本节开始，以后的锁、解锁中断都修改为对所有中断的锁和解锁。

程序在锁中断、解锁中断时会发生嵌套的现象，看下面的例子：

```
00001  MDS_IntLock
00002  ……
00003      MDS_IntLock
00004      ……
00005      MDS_IntUnlock
00006  ……
00007  MDS_IntUnlock
```

1 行和 7 行的锁中断和解锁中断是一对的，其目的是为了保护 2~6 行的代码。3 行和 5 行的锁中断和解锁中断是一对的，其目的是为了保护 4 行的代码。但这样带来一个问题，在 1 行和 3 行连续锁了 2 次中断，4 行的代码时已经处于锁中断状态，这是符合设计要求的，但在 5 行时解锁了一次中断，那么 6 行的代码就无法得到锁中断的保护了，这与设计是不符的。为了解决这个问题，在这两个函数里面做了计数统计，只有在未锁中断的状态下调用锁中断函数 MDS_IntLock 才会真正的去操作硬件寄存器，执行锁中断操作，在已锁中断的状态下调用锁中断函数 MDS_IntLock 只会增加其内部的变量计数，不对硬件寄存器做任何操作，做一个虚假的锁中断操作。同理，只有在已锁中断的状态下，并且变量计数为 1 的情况下调用解锁中断函数 MDS_IntUnlock 才会真正的去操作硬件寄存器，执行解锁中断操作，在其它情况下调用解锁中断函数 MDS_IntLock 只会减少其内部的变量计数，不对硬件做任何操作。

	函数内部操作	函数内部变量值
初始状态		0
MDS_IntLock	操作硬件寄存器，锁中断。变量加 1。	1
MDS_IntLock	变量加 1。	2
MDS_IntUnlock	变量减 1。	1
MDS_IntLock	变量加 1。	2
MDS_IntUnlock	变量减 1。	1
MDS_IntUnlock	操作硬件寄存器，解锁中断。变量减 1。	0

表 8　锁中断解锁中断函数内部状态变化

锁中断函数 MDS_IntLock 和解锁中断函数 MDS_IntUnlock 需要成对使用，这里所说的成对，不是代码编写上的成对，而是代码运行时的成对。

本节增加了 delay 表，我们需要再改造一下 TCB 结构，为任务设计一个挂接到 delay 表的链表节点结构，并且需要增加一个变量用来保存任务 delay 的时间，来看看新的 TCB 结构：

```
typedef struct m_tcb
{
    STACKREG strStackReg;           /* 备份寄存器组 */
    M_TCBQUE strTcbQue;             /* TCB 结构队列 */
    M_TCBQUE strDelayQue;           /* delay 表队列 */
    U32 uiTaskFlag;                 /* 任务标志 */
    U8 ucTaskPrio;                  /* 任务优先级 */
    M_TASKOPT strTaskOpt;           /* 任务参数 */
```

```
    U32 uiStillTick;                /* 延迟到的时间 */
}M_TCB;
```

strDelayQue 是任务挂入 delay 链表的队列结构。

任务是否在 delay 链表中的标志存放在 uiTaskFlag 变量中，uiTaskFlag 变量在后面的章节中还会扩展，增加其它的标志。

TCB 中还有一个 M_TASKOPT 结构，M_TASKOPT 结构如下：

```
typedef struct m_taskopt            /* 任务参数 */
{
    U8 ucTaskSta;                   /* 任务运行状态 */
    U32 uiDelayTick;                /* 延迟时间 */
}M_TASKOPT;
```

从本节开始，在创建任务时可以由用户指定任务创建时的状态。创建任务时，用户将任务的状态存入 M_TASKOPT 结构体中的 ucTaskSta 变量中，若是 delay 状态，则还需要将 delay 的 tick 数值存入 uiDelayTick 变量中，创建任务时将装有初始化数据的 M_TASKOPT 结构的变量的指针作为入口参数传递给 MDS_TaskCreate 函数，就可以指定任务创建时的状态了。

TCB 中的 uiStillTick 变量存放的是任务 delay 状态耗尽时的 tick 值，比如当前的 tick 是 100，任务需要 delay 5 个 ticks，那么该变量里保存的就是 105，表明该任务的 delay 态持续到 105 ticks。任务调度时就是根据该变量判断处于 delay 状态的任务是否需要转换为 ready 状态。

上面介绍了本小节新增的主要内容，比较零散，在介绍代码前，我们将这些内容串起来，梳理一下操作系统运行的过程。

整个系统上电后，在 MDS_SystemVarInit 函数里初始化操作系统的变量，然后创建最高优先级任务 MDS_RootTask，创建最低优先级任务 MDS_IdleTask，使用 MDS_TaskStart 函数从非操作系统状态切换到操作系统状态，开始运行 MDS_RootTask 任务。用户需要在 MDS_RootTask 任务里编写用户代码，对用户代码初始化，并初始化单板硬件，包括 tick 中断，创建用户任务。MDS_RootTask 任务具有最高优先级，为了让用户任务能得以运行，本节在 MDS_RootTask 任务最后调用 MDS_TaskDelay 函数，并使用 DELAYWAITFEV 参数，使 MDS_RootTask 任务永远处于 delay 状态。

当用户完成 tick 中断初始化之后，Mindows 的调度就正式开始了，此后每个 tick 周期就发生一次任务调度。若在 MDS_RootTask 任务运行过程中发生了 tick 调度，则由于 MDS_RootTask 任务具有最高优先级，tick 调度后还会继续运行 MDS_RootTask 任务。用户在 MDS_RootTask 任务中创建的任务可能处于 ready 态或者 delay 态，在 MDS_RootTask 任务处于 delay 态之后，用户任务就会在 tick 中断中开始调度，tick 中断调度发生时先查找 delay 表，从 delay 表中拆除 delay 时间耗尽的任务，将它们添加到 ready 表中，然后再在 ready 表中查找最高优先级的任务，切换到这个最高优先级任务继续运行。这个调度过程每个 tick 调度周期就会发生一次，由定时器周期触发。在任务运行过程中，如果调用了 MDS_TaskDelay 函数，任务就会进入 delay 态，被从 ready 表拆除，若不是永久 delay 则挂入 delay 表。MDS_TaskDelay 函数最后会调用软中断调度函数 MDS_TaskSwiSched 产生一次软中断调度，在软中断调度中不需要调度 delay 表，直接从 ready 表中找出最高优先级任务，切换到这个最高优先级任务继续运行，这个调度过程由实时事件触发，不定周期。Mindows 的调度方式就由这 2 种调度方式构成。在调度过程中，除 idle 任务之外的所有任务若都处于 delay 状态，操作系统则运行 idle 任务，当其它任务重新恢复到 ready 态时，它们就会抢占 idle 任务继续运行。

原理性的知识介绍完了，我们来看一下代码，先来看 MDS_TaskCreate 函数做了哪些改动：

```
00016  M_TCB* MDS_TaskCreate(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize,
00017                        U8 ucTaskPrio, M_TASKOPT* pstrTaskOpt)
00018  {
00019      M_TCB* pstrTcb;
00020
00021      /* pstrTaskOpt 参数并非必须，不做检查 */
00022
00023      /* 对创建任务所使用函数的指针合法性进行检查 */
00024      if(NULL == vfFuncPointer)
00025      {
00026          /* 指针为空，返回失败 */
00027          return (M_TCB*)NULL;
00028      }
00029
00030      /* 对任务堆栈合法性进行检查 */
00031      if((NULL == pucTaskStack) || (0 == uiStackSize))
00032      {
00033          /* 堆栈不合法，返回失败 */
00034          return (M_TCB*)NULL;
00035      }
00036
00037      /* 配置任务参数时对任务状态合法性进行检查 */
00038      if(NULL != pstrTaskOpt)
00039      {
00040          /* 不存在的任务状态返回失败 */
00041          if(!((TASKREADY == pstrTaskOpt->ucTaskSta)
00042               || (TASKDELAY == pstrTaskOpt->ucTaskSta)))
00043          {
00044              return (M_TCB*)NULL;
00045          }
00046      }
00047
00048      /* 对任务优先级检查，任务不能低于最低优先级 */
00049      if(ucTaskPrio > LOWESTPRIO)
00050      {
00051          return (M_TCB*)NULL;
00052      }
00053
00054      /* 对非系统任务优先级检查 */
00055      if((MDS_RootTask != vfFuncPointer) && (MDS_IdleTask != vfFuncPointer))
00056      {
00057          /* 非系统任务优先级不能是最高优先级，也不能低于最低优先级 */
00058          if((HIGHESTPRIO == ucTaskPrio) || (ucTaskPrio >= LOWESTPRIO))
00059          {
00060              return (M_TCB*)NULL;
00061          }
00062      }
00063
00064      /* 对是否重复创建 root 任务进行检查 */
00065      if(MDS_RootTask == vfFuncPointer)
00066      {
00067          /* root 任务已经创建过，返回失败 */
00068          if(NULL != gpstrRootTaskTcb)
00069          {
00070              return (M_TCB*)NULL;
00071          }
```

```
00072      }
00073
00074      /* 对是否重复创建 idle 任务进行检查 */
00075      if(MDS_IdleTask == vfFuncPointer)
00076      {
00077          /* idle 任务已经创建过，返回失败 */
00078          if(NULL != gpstrIdleTaskTcb)
00079          {
00080              return (M_TCB*)NULL;
00081          }
00082      }
00083
00084      /* 初始化 TCB */
00085      pstrTcb = MDS_TaskTcbInit(vfFuncPointer, pucTaskStack, uiStackSize, ucTaskPrio,
00086                                pstrTaskOpt);
00087
00088      /* 创建非系统任务后的操作 */
00089      if((MDS_RootTask != vfFuncPointer) && (MDS_IdleTask != vfFuncPointer))
00090      {
00091          /* 使用软中断调度任务 */
00092          MDS_TaskSwiSched();
00093      }
00094
00095      return pstrTcb;
00096  }
```

00017 行，新增入口参数 pstrTaskOpt 指针，通过 pstrTaskOpt 指针可以配置任务刚建立时的状态。若没有使用 pstrTaskOpt 参数，任务则被默认为 ready 态，若使用了 pstrTaskOpt 参数，其中的 ucTaskSta 变量中保存的是任务创建时的状态，uiDelayTick 变量中保存的是 delay 状态需要延迟的时间。

00038~00046 行，对 pstrTaskOpt 入口参数进行检查，若使用了此参数，则只能配置任务的初始状态为 ready 或者 delay 状态，否则返回失败。

00048~00062 行，对任务优先级进行检查，新创建任务的优先级不能低于任务最低优先级，若不是系统任务则不能使用最高和最低优先级。

00074~00082 行，对是否重复创建 idle 任务进行检查。

00089~00093 行，若创建非系统任务，则使用软中断调度一次任务，使新建立的这个任务也立刻参与到任务调度中。

MDS_TaskTcbInit 函数也增加了 pstrTaskOpt 入口参数，做了一些修改：

```
00108  M_TCB* MDS_TaskTcbInit(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize,
00109                         U8 ucTaskPrio, M_TASKOPT* pstrTaskOpt)
00110  {
00111      M_TCB* pstrTcb;
00112      M_CHAIN* pstrChain;
00113      M_CHAIN* pstrNode;
00114      M_PRIOFLAG* pstrPrioFlag;
00115      U8* pucStackBy4;
00116
00117      /* 堆栈满地址，需要 4 字节对齐 */
00118      pucStackBy4 = (U8*)(((U32)pucTaskStack + uiStackSize) & 0xFFFFFFFC);
00119
00120      /* TCB 结构存放的地址，需要 4 字节对齐 */
00121      pstrTcb = (M_TCB*)(((U32)pucStackBy4 - sizeof(M_TCB)) & 0xFFFFFFFC);
00122
```

```
00123       /* 初始化任务堆栈 */
00124       MDS_TaskStackInit(pstrTcb, vfFuncPointer);
00125
00126       /* 先将任务标志初始化为全空，后面再为其增加具体的功能标志 */
00127       pstrTcb->uiTaskFlag = 0;
00128
00129       /* 初始化指向 TCB 的指针 */
00130       pstrTcb->strTcbQue.pstrTcb = pstrTcb;
00131       pstrTcb->strDelayQue.pstrTcb = pstrTcb;
00132
00133       /* 初始化任务优先级 */
00134       pstrTcb->ucTaskPrio = ucTaskPrio;
00135
00136       /* 没有任务参数则将任务状态设置为 ready 态 */
00137       if(NULL == pstrTaskOpt)
00138       {
00139           pstrTcb->strTaskOpt.ucTaskSta = TASKREADY;
00140       }
00141       else /* 有任务参数则将参数复制到 TCB 中 */
00142       {
00143           pstrTcb->strTaskOpt.ucTaskSta = pstrTaskOpt->ucTaskSta;
00144           pstrTcb->strTaskOpt.uiDelayTick = pstrTaskOpt->uiDelayTick;
00145       }
00146
00147       /* 锁中断，防止其它任务影响 */
00148       (void)MDS_IntLock();
00149
00150       /* 建立的任务包含 ready 态，将任务加入 ready 表 */
00151       if(TASKREADY == (TASKREADY & pstrTcb->strTaskOpt.ucTaskSta))
00152       {
00153           pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00154           pstrNode = &pstrTcb->strTcbQue.strQueHead;
00155           pstrPrioFlag = &gstrReadyTab.strFlag;
00156
00157           /* 将该任务添加到 ready 表中 */
00158           MDS_TaskAddToSchedTab(pstrChain, pstrNode, pstrPrioFlag, ucTaskPrio);
00159       }
00160
00161       /* 建立的任务包含 delay 态，将任务加入 delay 表 */
00162       if(TASKDELAY == (TASKDELAY & pstrTcb->strTaskOpt.ucTaskSta))
00163       {
00164           /* 非永久等待任务才挂入 delay 表 */
00165           if(DELAYWAITFEV != pstrTaskOpt->uiDelayTick)
00166           {
00167               /* 更新新建任务的延迟时间 */
00168               pstrTcb->uiStillTick = guiTick + pstrTaskOpt->uiDelayTick;
00169
00170               /* 从任务参数里获取 delay 表节点并加入到 delay 表 */
00171               pstrNode = &pstrTcb->strDelayQue.strQueHead;
00172               MDS_TaskAddToDelayTab(pstrNode);
00173
00174               /* 置任务在 delay 表标志 */
00175               pstrTcb->uiTaskFlag |= DELAYQUEFLAG;
00176           }
00177       }
00178
00179       /* 挂入链表后解锁中断，允许任务调度 */
00180       (void)MDS_IntUnlock();
00181
```

```
00182       return pstrTcb;
00183   }
```

00127 行，初始化任务标志为空，即没有任何标志的状态，后续需要改变标志状态时再处理。

00131 行，初始化 strDelayQue 结构中的 TCB 指针。

00137~00140 行，创建任务时若没有使用 pstrTaskOpt 任务参数，则创建的任务默认为 ready 态。

00142~00145 行，使用了 pstrTaskOpt 任务参数，将任务参数复制到 TCB 中。

00162 行，判断新创建的任务是否为 delay 状态。

00165 行，判断新创建的任务是否为非永久 delay 状态。

00167 行，对于非永久 delay 任务，将需要 delay 的 tick 数值换算为 delay 时间耗尽时的 tick 数值。

00171 行，获取任务 TCB 中可挂入 delay 链表的节点。

00172 行，将任务挂入到 delay 链表。

00175 行，在任务标志 uiTaskFlag 中设置任务已挂入到 delay 链表的标志。

MDS_TaskAddToDelayTab 函数的功能是将任务节点添加到 delay 表中，添加的时候是按照任务剩余时 delay 时间从少到多排序的，这个函数最关键的部分在于为新加入的节点找到合适的节点位置，需要使用新加入节点的 delay 耗尽 tick 数值依次与 delay 表中节点的 delay 耗尽 tick 数值以及当前的 tick 数值做比较。当前 tick 变量 guiTick 会从 0 开始递增，当到达最大值 $2^{32}-1$ 时又会重新回到 0，形成一个回环的计数过程，因此，这 3 个需要比较的数值存在多种组合情况，情况比较复杂，MDS_TaskAddToDelayTab 函数的细节不再详细介绍了。

下面来看一下 MDS_TaskDelay 函数的代码：

```
00196   U32 MDS_TaskDelay(U32 uiDelayTick)
00197   {
00198       M_CHAIN* pstrChain;
00199       M_CHAIN* pstrNode;
00200       M_CHAIN* pstrDelayNode;
00201       M_TCBQUE* pstrTaskQue;
00202       M_PRIOFLAG* pstrPrioFlag;
00203       U8 ucTaskPrio;
00204
00205       /* idle 任务不能处于 delay 状态 */
00206       if(gpstrCurTcb == gpstrIdleTaskTcb)
00207       {
00208           return RTN_FAIL;
00209       }
00210
00211       /* 延迟时间不为 0 tick 则调度任务，否则不延迟直接切换任务 */
00212       if(DELAYNOWAIT != uiDelayTick)
00213       {
00214           /* 获取当前任务的相关调度参数 */
00215           ucTaskPrio = gpstrCurTcb->ucTaskPrio;
00216           pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00217           pstrPrioFlag = &gstrReadyTab.strFlag;
00218
00219           (void)MDS_IntLock();
00220
00221           /* 将当前任务从 ready 表拆除 */
```

```
00222           pstrNode = MDS_TaskDelFromSchedTab(pstrChain, pstrPrioFlag, ucTaskPrio);
00223
00224           /* 清除任务的 ready 状态 */
00225           gpstrCurTcb->strTaskOpt.ucTaskSta &= ~((U8)TASKREADY);
00226
00227           /* 更新当前任务的延迟时间 */
00228           gpstrCurTcb->strTaskOpt.uiDelayTick = uiDelayTick;
00229
00230           /* 非永久等待任务才挂入 delay 表 */
00231           if(DELAYWAITFEV != uiDelayTick)
00232           {
00233               gpstrCurTcb->uiStillTick = guiTick + uiDelayTick;
00234
00235               /* 获取当前任务 delay 表的任务节点 */
00236               pstrTaskQue = (M_TCBQUE*)pstrNode;
00237               pstrDelayNode = &pstrTaskQue->pstrTcb->strDelayQue.strQueHead;
00238
00239               /* 将当前任务加入到 delay 表 */
00240               MDS_TaskAddToDelayTab(pstrDelayNode);
00241
00242               /* 置任务在 delay 表标志 */
00243               gpstrCurTcb->uiTaskFlag |= DELAYQUEFLAG;
00244           }
00245
00246           /* 增加任务的 delay 状态 */
00247           gpstrCurTcb->strTaskOpt.ucTaskSta |= TASKDELAY;
00248
00249           (void)MDS_IntUnlock();
00250       }
00251       else /* 任务不延迟，仅发生任务切换 */
00252       {
00253           /* 借用 uiDelayTick 变量保存延迟任务的返回值 */
00254           gpstrCurTcb->strTaskOpt.uiDelayTick = RTN_SUCD;
00255       }
00256
00257       /* 使用软中断调度任务 */
00258       MDS_TaskSwiSched();
00259
00260       /* 返回延迟任务的返回值，任务从 delay 状态返回时返回值被保存在 uiDelayTick 中 */
00261       return gpstrCurTcb->strTaskOpt.uiDelayTick;
00262   }
```

00196 行，函数返回值分 4 种，RTN_SUCD: 任务没有 delay，仅发生任务切换，仅在入口参数为 0 时才会返回该值。RTN_FAIL: 任务 delay 失败，没有进入 delay 状态。RTN_TKDLTO: 任务已经进入过 delay 状态，并且又从 delay 状态返回到 running 状态，delay 的时间已耗尽，超时返回。RTN_TKDLBK: 任务 delay 状态被打断，被其它任务使用 MDS_TaskWake 函数唤醒。

入口参数 uiDelayTick 是需要 delay 的 tick 数。

00206~00209 行，idle 任务不能延迟。

00212 行，对任务 delay 的时间进行判断，不是 delay 0 tick 的情况走下面分支。

00222 行，当前任务需要切换到 delay 状态，从 ready 表拆除。

00225 行，清除任务的 ready 状态。

00227 行，更新当前任务的延迟时间。

00231 行，对任务 delay 的时间进行判断，不是永久 delay 的情况走下面分支。

00233 行，计算该任务需要 delay 到的 tick 数值，存入 TCB 中。

00236 行，将当前任务的 M_TCBQUE 型 ready 节点指针强制转换为 M_TCBQUE 型指针。

00237 行，从任务的 ready 节点找到 delay 节点指针。

00240 行，将该任务加入 delay 表。

00243 行，在该任务的 TCB 中的任务标志 uiTaskFlag 中设置标志，表明该任务已经处于 delay 表中。

00247 行，增加任务的 delay 状态。

00251~00255 行，delay 0 tick 时走此分支，任务只切换不延迟，将返回值保存在 strTaskOpt 中的 uiDelayTick 变量中。

00258 行，任务相关变量、ready 表、delay 表的操作已经完成，调用 MDS_TaskSwiSched 函数触发软中断，开始任务调度。

00263 行，将 strTaskOpt 中的 uiDelayTick 变量作为返回值返回给上级父函数。uiDelayTick 变量中的返回值会在 248 行仅发生任务切换时存入，或者在任务调度函数 MDS_TaskDelayTabSched 中 delay 时间耗尽时存入，或者在 MDS_TaskWake 函数中由其它任务唤醒时存入。

261 行与 258 行在代码编写上是连在一起的，但在执行时可能会有时间间隔，258 行会发生任务调度，中间可能会插入其它任务的执行过程。

MDS_TaskWake 函数正好与 MDS_TaskDelay 函数相反，它将处于 delay 状态的任务从 delay 表拆除，添加到 ready 表，并修改任务的一些相关状态变量，代码如下，不再详细介绍。

```
00270  U32 MDS_TaskWake(M_TCB* pstrTcb)
00271  {
00272      M_CHAIN* pstrChain;
00273      M_CHAIN* pstrNode;
00274      M_PRIOFLAG* pstrPrioFlag;
00275      U8 ucTaskPrio;
00276
00277      /* 入口参数检查 */
00278      if(NULL == pstrTcb)
00279      {
00280          return RTN_FAIL;
00281      }
00282
00283      (void)MDS_IntLock();
00284
00285      /* 仅可以唤醒任务的 delay 状态 */
00286      if(TASKDELAY != (TASKDELAY & pstrTcb->strTaskOpt.ucTaskSta))
00287      {
00288          (void)MDS_IntUnlock();
00289
00290          return RTN_FAIL;
00291      }
00292
00293      pstrNode = &pstrTcb->strDelayQue.strQueHead;
00294
00295      /* 非永久等待任务才从 delay 表拆除 */
00296      if(DELAYWAITFEV != pstrTcb->strTaskOpt.uiDelayTick)
00297      {
00298          /* 从 delay 表拆除该任务 */
00299          (void)MDS_ChainCurNodeDelete(&gstrDelayTab, pstrNode);
```

```
00300
00301           /* 置任务不在 delay 表标志 */
00302           pstrTcb->uiTaskFlag &= (~((U32)DELAYQUEFLAG));
00303       }
00304
00305       /* 清除任务的 delay 状态 */
00306       pstrTcb->strTaskOpt.ucTaskSta &= ~((U8)TASKDELAY);
00307
00308       /* 借用 uiDelayTick 变量保存延迟任务的返回值 */
00309       pstrTcb->strTaskOpt.uiDelayTick = RTN_TKDLBK;
00310
00311       /* 获取该任务的相关参数 */
00312       ucTaskPrio = pstrTcb->ucTaskPrio;
00313       pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00314       pstrPrioFlag = &gstrReadyTab.strFlag;
00315
00316       /* 将该任务添加到 ready 表中 */
00317       MDS_TaskAddToSchedTab(pstrChain, pstrNode, pstrPrioFlag, ucTaskPrio);
00318
00319       /* 增加任务的 ready 状态 */
00320       pstrTcb->strTaskOpt.ucTaskSta |= TASKREADY;
00321
00322       (void)MDS_IntUnlock();
00323
00324       /* 使用软中断调度任务 */
00325       MDS_TaskSwiSched();
00326
00327       return RTN_SUCD;
00328   }
```

现在我们已经拥有了 ready 和 delay 两种任务状态，那么在 tick 中断里就需要对这两个调度表进行操作。这个过程分为 2 步，先对 delay 表操作，将 delay 时间耗尽的任务从 delay 表拆除，挂入 ready 表，使之能参与到 ready 表的调度之中，然后再对 ready 表进行调度。

来看调度函数的代码：

```
00345   void MDS_TaskSched(void)
00346   {
00347       M_TCB* pstrTcb;
00348
00349       /* 调度 delay 表任务 */
00350       MDS_TaskDelayTabSched();
00351
00352       /* 调度 ready 表任务 */
00353       pstrTcb = MDS_TaskReadyTabSched();
00354
00355       /* 准备任务切换 */
00356       MDS_TaskSwitch(pstrTcb);
00357   }
```

有关 ready 表调度的代码被封装到了 353 行的 MDS_TaskReadyTabSched 函数里面，本节新增的 delay 表调度代码被封装到 350 行的 MDS_TaskDelayTabSched 函数里面，下面来看看 MDS_TaskDelayTabSched 函数的代码：

```
00399   void MDS_TaskDelayTabSched(void)
00400   {
```

```
00401       M_TCB* pstrTcb;
00402       M_CHAIN* pstrChain;
00403       M_CHAIN* pstrNode;
00404       M_CHAIN* pstrDelayNode;
00405       M_CHAIN* pstrNextNode;
00406       M_PRIOFLAG* pstrPrioFlag;
00407       M_TCBQUE* pstrDelayQue;
00408       U32 uiTick;
00409       U8 ucTaskPrio;
00410
00411       /* 获取 delay 表中的任务节点 */
00412       pstrDelayNode = MDS_ChainEmpInq(&gstrDelayTab);
00413
00414       /* delay 表中有任务, 调度 delay 表中的任务 */
00415       if(NULL != pstrDelayNode)
00416       {
00417           /* 判断 delay 表中任务的延迟时间是否结束 */
00418           while(1)
00419           {
00420               /* 获取 delay 表中的任务的延迟时间 */
00421               pstrDelayQue = (M_TCBQUE*)pstrDelayNode;
00422               pstrTcb = pstrDelayQue->pstrTcb;
00423               uiTick = pstrTcb->uiStillTick;
00424
00425               /* 该任务延迟时间到, 从 delay 表中删除并加入到调度表中 */
00426               if(uiTick == guiTick)
00427               {
00428                   /* 从 delay 表拆除该任务 */
00429                   pstrNextNode = MDS_ChainCurNodeDelete(&gstrDelayTab, pstrDelayNode);
00430
00431                   /* 置任务不在 delay 表标志 */
00432                   pstrTcb->uiTaskFlag &= (~((U32)DELAYQUEFLAG));
00433
00434                   /* 清除任务的 delay 状态 */
00435                   pstrTcb->strTaskOpt.ucTaskSta &= ~((U8)TASKDELAY);
00436
00437                   /* 借用 uiDelayTick 变量保存 delay 任务的返回值 */
00438                   pstrTcb->strTaskOpt.uiDelayTick = RTN_TKDLTO;
00439
00440                   /* 获取该任务的相关参数 */
00441                   pstrNode = &pstrTcb->strTcbQue.strQueHead;
00442                   ucTaskPrio = pstrTcb->ucTaskPrio;
00443                   pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00444                   pstrPrioFlag = &gstrReadyTab.strFlag;
00445
00446                   /* 将该任务添加到 ready 表中 */
00447                   MDS_TaskAddToSchedTab(pstrChain, pstrNode, pstrPrioFlag,
00448                                         ucTaskPrio);
00449
00450                   /* 增加任务的 ready 状态 */
00451                   pstrTcb->strTaskOpt.ucTaskSta |= TASKREADY;
00452
00453                   /* delay 表已经调度完毕, 结束对 delay 表的调度 */
00454                   if(NULL == pstrNextNode)
00455                   {
00456                       break;
00457                   }
00458                   else /* delay 表没调度完, 更新下个节点继续判断 */
00459                   {
```

```
00460                       pstrDelayNode = pstrNextNode;
00461                   }
00462               }
00463               else /* 所有任务都没有到时间，结束对 delay 表的调度 */
00464               {
00465                   break;
00466               }
00467           }
00468       }
00469   }
```

00412 行，从 delay 表中获取第一个子节点 pstrDelayNode。

00415 行，判断第一个子节点是否为空。

00418 行，循环查询 delay 表中的任务。执行到此行说明 delay 链表不为空，说明有处于 delay 状态的任务。

000421 行，将 M_CHAIN 型的指针 pstrDelayNode 强制转换为 M_TCBQUE 型的指针 pstrDelayQue。

00422 行，通过 pstrDelayQue 获取 TCB 指针。

00423 行，通过 TCB 指针获取到任务 delay 状态耗尽时的 tick 数值。

00426 行，被查询的任务 delay 状态在当前 tick 已经耗尽，需要从 delay 状态转换为 ready 状态，走下面分支。

00429 行，将该任务从 delay 表中拆除。

00432 行，从 TCB 中的任务标志 uiTaskFlag 中去除任务处于 delay 表的标志。

00435 行，去除任务的 delay 状态。

00438 行，将 delay 耗尽的返回值存入 strTaskOpt 中的 uiDelayTick 变量，供 MDS_TaskDelay 函数返回时使用。

00441~00451 行，将该任务添加到 ready 表中。

00454~00461 行，若下个节点为空，说明 delay 表已经遍历完毕，退出调度函数。若下个节点不为空则继续查询。

00463~00466 行，走到此分支说明当前节点 delay 时间还没有耗尽，因为 delay 表是按照 delay 耗尽时间从少到多的顺序排列的，因此后面节点的时间也不会耗尽，不需要再遍历，直接返回。

下面介绍一下与软中断相关的函数。

Mindows 中使用 MDS_TaskSwiSched 函数触发软中断，MDS_TaskSwiSched 函数里面使用 MDS_RunInInt 函数对芯片当前的状态做了一个判断，避免在中断中触发软中断造成异常，实际触发软中断的函数是 MDS_TaskOccurSwi 函数。

```
00137  void MDS_TaskSwiSched(void)
00138  {
00139      /* 如果在中断中运行该函数则直接返回 */
00140      if(RTN_SUCD == MDS_RunInInt())
00141      {
00142          return;
00143      }
00144
00145      /* 调用软中断，在软中断服务程序中调度任务 */
00146      MDS_TaskOccurSwi(SWI_TASKSCHED);
00147  }
```

MDS_RunInInt 调用了汇编函数 MDS_GetChipMode 获取芯片的工作模式，若不是 USR 模式则表明当前是在中断中运行。

```
00155  U32 MDS_RunInInt(void)
00156  {
00157      /* 当前程序在中断中运行 */
00158      if(MODE_USR != MDS_GetChipMode())
00159      {
00160          return RTN_SUCD;
00161      }
00162      else
00163      {
00164          return RTN_FAIL;
00165      }
00166  }
```

MDS_GetChipMode 函数是一个汇编函数，CPSR 寄存器中的 bit4~bit0 中保存的就是当前模式，它将 CPSR 寄存器的 bit4~bit0 作为函数返回值保存到 R0 中。

```
00114      .func MDS_GetChipMode
00115  MDS_GetChipMode:
00116
00117      MRS    R0, CPSR
00118      AND    R0, R0, #0x1F
00119      BX     R14
00120
00121      .endfunc
```

MDS_TaskOccurSwi 函数也是一个汇编函数，它有一个入口参数，用来表明该软中断的功能，C 语言的函数原型为

```
MDS_TaskOccurSwi(U32 uiSwiNo);
```

汇编语言的源代码为：

```
00127      .func MDS_TaskOccurSwi
00128  MDS_TaskOccurSwi:
00129
00130      SWI    0
00131      BX     R14
00132
00133      .endfunc
```

00130 行，执行“SWI”汇编指令，该指令会触发软中断，将 PC 指针跳转到软中断向量表。在 startup.s 文件的软中断向量表里面使用 MDS_SwiContextSwitch 函数作为软中断的服务函数，它的功能与 tick 中断的 MDS_TickContextSwitch 函数功能非常相似，都是用来调度任务的，但还是有一些不同的地方。

```
00054      .func MDS_SwiContextSwitch
00055  MDS_SwiContextSwitch:
00056
00057      @保存接口寄存器
00058      ADD    R14, R14, #4
00059      STMDB  R13!, {R0 - R3, R12, R14}
00060
00061      @调用 C 语言 SWI 中断处理函数
```

```
00062       LDR     R3, =MDS_SwiIsr
00063       MOV     R14, PC
00064       BX      R3
00065
00066       @保存当前任务的堆栈信息
00067       LDR     R0, =gpstrCurTaskSpAddr
00068       LDR     R14, [R0]
00069       MRS     R0, SPSR
00070       STMIA   R14!, {R0}
00071       LDMIA   R13!, {R0 - R3, R12}
00072       STMIA   R14, {R0 - R14}^
00073       ADD     R14, R14, #0x3C
00074       LDMIA   R13!, {R0}
00075       STMIA   R14, {R0}
00076
00077       @任务调度完毕，恢复将要运行任务现场
00078       LDR     R0, =gpstrNextTaskSpAddr
00079       LDR     R14, [R0]
00080       LDMIA   R14, {R0}
00081       MSR     SPSR, R0
00082       LDMIB   R14, {R0 - R14}^
00083       NOP
00084       ADD     R14, R14, #0x40
00085       LDMIA   R14, {R14}
00086       SUB     R14, R14, #4
00087       MOVS    PC, R14
00088
00089       .endfunc
```

00058行，将返回的LR寄存器地址多加4字节，这是因为在进入MDS_TickContextSwitch函数时硬件会自动多加4字节，为了使MDS_SwiContextSwitch函数与MDS_TickContextSwitch函数的栈格式相同，这里先多加4字节，在MDS_SwiContextSwitch函数返回前会再多减去4字节。

00062~00064行，调用软中断处理函数MDS_SwiIsr。注意，这里不能使用R0寄存器，因为此时R0中保存的是MDS_TaskOccurSwi函数传递进来的入口参数。

00067~00085行，寄存器组备份还原过程，与MDS_TickContextSwitch函数完全一致。

00086行，函数返回前减去前面多加的4字节。

00087行，软中断调度完成，使用MOVS指令返回，这点与tick中断所使用的ISR中断不同。

MDS_SwiIsr函数才是真正的软中断处理过程，使用C语言编写，所有的软中断处理过程都在这个函数内实现。

```
00094  void MDS_SwiIsr(U32 uiSwiNo)
00095  {
00096      /* 软中断产生的任务调度 */
00097      if(SWI_TASKSCHED == (SWI_TASKSCHED & uiSwiNo))
00098      {
00099          /* 调度任务 */
00100          MDS_TaskReadySched();
00101      }
00102      else /* 中断源是非任务调度中断 */
00103      {
00104          /* 非任务调度中断引发的中断里需要更新一次任务调度变量 */
00105          MDS_TaskSwitch(gpstrCurTcb);
```

```
00106
00107           switch(uiSwiNo)
00108           {
00109               case SWI_INTENABLE:     /* 中断使能 */
00110
00111                   MDS_IntEnable();
00112
00113                   break;
00114
00115               case SWI_INTDISABLE:    /* 中断禁止 */
00116
00117                   MDS_IntDisable();
00118
00119                   break;
00120
00121               /*************** 用户可以在下面区域添加自定义软中断代码 ***************/
00122
00123               default:
00124
00125                   break;
00126
00127               /*************** 用户可以在上面区域添加自定义软中断代码 ***************/
00128           }
00129       }
00130   }
```

00094 行，入口参数 uiSwiNo 从 MDS_TaskOccurSwi 函数开始，被保存在 R0 中，经历了 MDS_SwiContextSwitch 函数传递给 MDS_SwiIsr 函数。

00097 行，软中断调度的中断走此分支。

00100 行，在软中断中调度任务，只需要调度 ready 表。delay 表的调度是以 tick 为周期的，在 tick 中断中才能调度。

00102 行，非软中断调度产生的软中断走此分支。

00105 行，更新调度中所使用的全局变量。尽管是非软中断调度，但由于该函数的上级函数 MDS_SwiContextSwitch 会对寄存器组进行备份、恢复，因此此处也需要更新一下调度中所使用的全局变量。

00107 行，选择不同的软中断命令分支。

00109~00113 行，解锁中断命令，调用 MDS_IntEnable 函数解锁中断。MDS_IntEnable 函数是汇编函数，直接修改 CPSR 中的全局中断控制位允许中断产生。

00115~00119 行，锁中断命令，调用 MDS_IntDisable 函数锁中断。MDS_IntDisable 函数是汇编函数，直接修改 CPSR 中的全局中断控制位禁止中断产生。

00121~00127 行，用户可以将自定义的软中断命令放在此处执行。

MDS_IntEnable 和 MDS_IntDisable 函数直接操作的对象是 SVC 模式下 $SPSR_{SVC}$ 寄存器，当程序从 SVC 模式返回到 USR 模式时，$SPSR_{SVC}$ 寄存器就会被恢复到 CPSR 中，达到修改 CPSR 寄存器控制中断产生的目的。

```
00139       .func MDS_IntEnable
00140   MDS_IntEnable:
00141
00142       MRS     R0, SPSR
00143       LDR     R1, =(CPSR_IRQENABLE & CPSR_FIQENABLE)
00144       AND     R0, R0, R1
00145       MSR     SPSR, R0
```

```
00146       BX      R14
00147
00148   .endfunc
```

```
00154       .func MDS_IntDisable
00155   MDS_IntDisable:
00156
00157       MRS     R0, SPSR
00158       LDR     R1, =(CPSR_IRQDISABLE | CPSR_FIQDISABLE)
00159       ORR     R0, R0, R1
00160       MSR     SPSR, R0
00161       BX      R14
00162
00163       .endfunc
```

提供给用户使用的锁中断函数是 MDS_IntLock，解锁中断函数是 MDS_IntUnlock，它们同软中断调度函数一样，都是通过调用 MDS_TaskOccurSwi 函数触发软中断，通过入口参数区分功能，在软中断服务函数 MDS_SwiIsr 里实现自己的功能。

```
00650   U32 MDS_IntLock(void)
00651   {
00652       /* 非操作系统状态，不设置中断 */
00653       if(NULL == gpstrCurTcb)
00654       {
00655           return RTN_SUCD;
00656       }
00657
00658       /* 如果在中断中运行该函数则直接返回 */
00659       if(RTN_SUCD == MDS_RunInInt())
00660       {
00661           return RTN_SUCD;
00662       }
00663
00664       /* 第一次调用该函数才做实际的锁中断操作 */
00665       if(0 == guiIntLockCounter)
00666       {
00667           MDS_TaskOccurSwi(SWI_INTDISABLE);
00668
00669           guiIntLockCounter++;
00670
00671           return RTN_SUCD;
00672       }
00673       /* 非第一次调用该函数并且小于最大次数则直接返回成功 */
00674       else if(guiIntLockCounter < 0xFFFFFFFF)
00675       {
00676           guiIntLockCounter++;
00677
00678           return RTN_SUCD;
00679       }
00680       else /* 超出最大次数则直接返回失败 */
00681       {
00682           return RTN_FAIL;
00683       }
00684   }
```

00653~00656 行，在还没有进入操作系统状态前不能执行此函数。

00659~00662 行，在中断中不能执行此函数。

00665~00672 行，guiIntLockCounter 变量里存放的是锁中断、解锁中断的次数，初始化为 0，当 guiIntLockCounter 为 0 时，说明是第一次执行该函数，只有第一次执行该函数时才操作硬件寄存器，执行锁中断动作。

需要注意一点，669 行 guiIntLockCounter 变量自加需要在 667 行 MDS_TaskOccurSwi 函数之后执行，因为 MDS_IntLock 函数可能会重入，这样做在函数重入时，不会使 guiIntLockCounter 变量计数产生错误。

00674~00679 行，guiIntLockCounter 变量没有溢出则做自加操作，返回锁中断成功，是一个虚假的锁中断过程。

00680~00683 行，guiIntLockCounter 变量溢出，返回成功。

```
00693  U32 MDS_IntUnlock(void)
00694  {
00695      /* 非操作系统状态，不设置中断 */
00696      if(NULL == gpstrCurTcb)
00697      {
00698          return RTN_SUCD;
00699      }
00700
00701      /* 如果在中断中运行该函数则直接返回 */
00702      if(RTN_SUCD == MDS_RunInInt())
00703      {
00704          return RTN_SUCD;
00705      }
00706
00707      /* 非第一次调用该函数直接返回成功 */
00708      if(guiIntLockCounter > 1)
00709      {
00710          guiIntLockCounter--;
00711
00712          return RTN_SUCD;
00713      }
00714      /* 最后一次调用该函数才做实际的解锁中断操作 */
00715      else if(1 == guiIntLockCounter)
00716      {
00717          guiIntLockCounter--;
00718
00719          MDS_TaskOccurSwi(SWI_INTENABLE);
00720
00721          return RTN_SUCD;
00722      }
00723      else /* 等于 0 次则直接返回失败 */
00724      {
00725          return RTN_FAIL;
00726      }
00727  }
```

MDS_IntUnlock 函数与 MDS_IntLock 函数，非常相似，只有当 guiIntLockCounter 变量为 1 时才操作硬件寄存器，执行解锁中断动作。

本节操作系统部分的内容就介绍到这里。

本节用户代码在打印输出方面做了较大的修改，从本节开始，串口打印功能不再由任务实时向串口打印，而是由任务先将字符串打印到内存，然后使用一个低优先级的任务从内存中取出字符串打印到串口。串口是一个低速率的外设，9600 的波特率差不多 1ms 打印 1 个

字符，打印几十个字符的字符串时就需要几十 ms，如果采用原有的打印方式，在打印字符串的过程中可能会发生任务切换，如果切换后的任务也在向串口打印字符串，那么这 2 个任务的字符就会混在一起，串口输出的字符发生混乱，得不到我们预期的输出结果。如果采用新的打印方式，每个任务需要从内存申请一个消息缓冲用来存放打印的字符，申请消息缓冲的过程非常快，可以将这一过程用中断锁住，防止其它任务干扰。消息缓冲对于任务来说，它是私有的，任务将字符串打印到消息缓冲的过程不会受任务切换的影响。当任务向消息缓冲打印完成后，先锁中断，然后将消息缓冲挂入消息队列，再解锁中断。消息打印任务会不断的查询消息队列，从队列中获取消息缓冲，将消息缓冲里面的数据打印到串口。向串口打印消息采用的是中断方式，消息打印任务将每条消息的第一个字符输出到串口，消息中剩下的其余字符会由上个字符触发的中断打印到串口，直至打印完最后一个字符。由于向串口打印数据是由同一个任务完成的，因此不会存在打印字符混乱的情况。

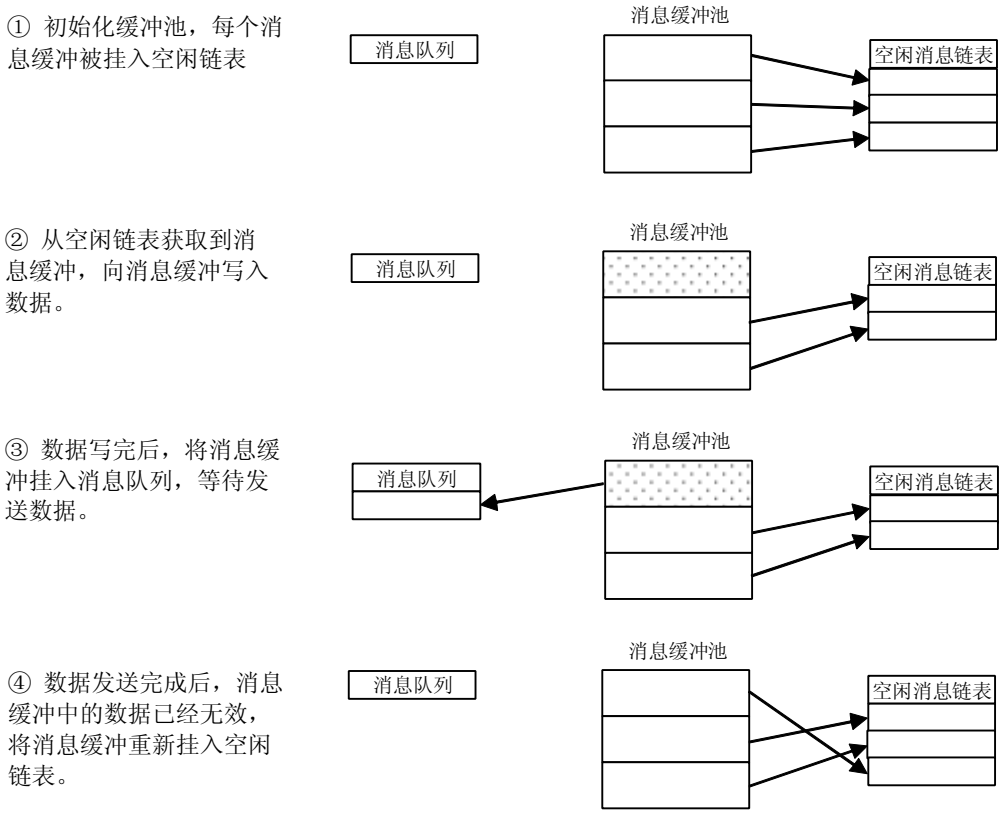


图 43　打印消息处理过程

我们可以定义一个消息缓冲池，所有的消息缓冲都在这里：

```
BUFPOOL gstrBufPool;
```

展开里面的结构体：

```
typedef struct bufpool
{
    M_CHAIN strFreeList;                /* 缓冲池空闲链表 */
    MSGBUF astrBufPool[BUFPOOLNUM];     /* 缓冲池 */
}BUFPOOL;

typedef struct msgbuf
{
    M_CHAIN strChain;                   /* 缓冲链表 */
```

```
    U8 ucLength;                        /* 消息长度 */
    U8 ucCounter;                       /* 消息收发计数 */
    U8 aucBuf[MSGBUFLEN];               /* 消息缓冲 */
}MSGBUF;
```

其中 strFreeList 是空闲消息链表，初始化消息缓冲池的时候将所有消息缓冲挂接到这个链表上，申请消息缓冲时从该链表就可以获取到空闲的消息缓冲。当消息缓冲使用完毕后需要再挂接到这个链表上，可以继续重复使用。strChain 是每个消息缓冲挂接到空闲消息链表的节点，ucLength 中存储的是消息的长度，ucCounter 中存储的是消息收发时的计数，aucBuf 是存放消息的数组。

任务向消息缓冲填充完字符串之后，需要将消息挂入队列，目前的队列结构就是一个链表，后续章节我们会再补充一些东西进来。

```
typedef struct m_que    /* 队列结构 */
{
    M_CHAIN strChain;   /* 队列链表 */
}M_QUE;
```

关于消息打印部分的细节就不做过多介绍了，请读者自行参考代码。

在学习 C 语言时，我们都使用过 printf 函数，这个函数可以根据个人需要,很灵活的向显示器终端打印数据，比如，我们可以按照下面的方式输出：

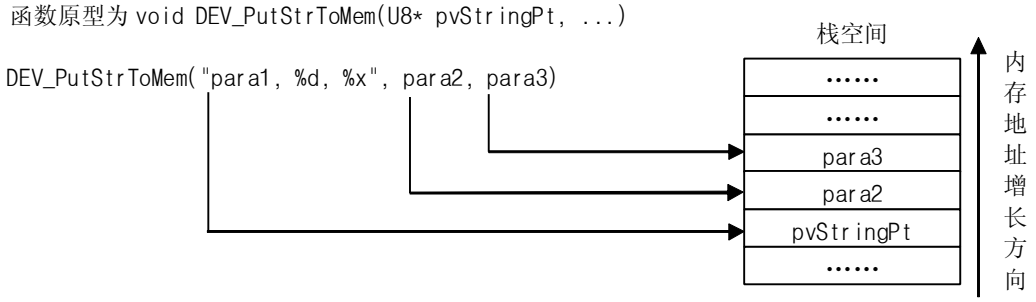
```
printf("Wanlix & Mindows");
printf("%d, %c", i, j);
```

不知道你注意过没有，printf 函数的参数个数是可变的，上面的第一个例子只有 1 个参数，第二个例子有 3 个参数，这点与我们一般所使用的函数是不同的。在 Mindows 中，我们仿造 printf 函数，将向内存打印的函数 DEV_PutStrToMem 也编写成一个参数可变的函数，下面我们一起来看看这是怎么实现的。

参数可变的函数原型定义为：

```
void DEV_PutStrToMem(U8* pvStringPt, ...);
```

其中比较特别的是“...”，这三个点表示省略的参数。可变参数函数的原理非常简单，可变参数函数的参数不像我们前面讲过的那样，使用 R0~R3 寄存器传递参数，而是直接使用堆栈传递参数，而且这些参数都是连在一起存放的，而函数原型中第一个参数是固定的，我们可以获取到第一个参数的地址&pvStringPt，然后将这个地址加 4 就可以得到第二个参数的地址，再加 4 就是第三个参数的地址，依次类推，这样就可以获取到任意多的参数地址了，有了参数地址那么获取参数的内容也就不成问题了。还有一个问题要解决，DEV_PutStrToMem 函数是怎么知道它究竟有多少个参数的呢？注意到参数里的“%”号了么，每个%号对应一个参数，我们只需要查找%号的个数就可以确定可变参数的个数，再将每个可变参数转换为对应的%号后面的格式就可以实现 DEV_PutStrToMem 函数的功能了。




```
假设字符串"para1, %d, %x"的地址为 addr,
addr = (U32)&pvStringPt;
para2 = *((U32*)(addr + 4));
para3 = *((U32*)(addr + 8));
```

图 44　可变参数函数

可变参数函数的原型可以有多种形式，但必须要保证：

1.必须有第一个参数，通过这个参数才能获取到参数存放在栈中的首地址，后面的参数才是可变的。

2.在设计可变参数时需要能体现出可变参数的个数，如上面查找第一个参数中%号的方法，或者将第一个参数定义为可变参数的个数，或者其它方式。

在 C 语言标准头文件 stdarg.h 里面已经为可变函数定义了几个宏，使用这些宏也可以实现可变参数函数，原理都一样， 细节不再介绍了。

本节内容已经介绍完毕，设计一下测试函数来验证本节新增加的功能。本节共有 4 个测试函数 TEST_TestTask1~TEST_TestTask4，每个函数的结构非常相似，循环执行打印、运行、延迟这 3 个过程。

```
00020  void TEST_TestTask1(void)
00021  {
00022      while(1)
00023      {
00024          DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00025                          MDS_SystemTickGet());
00026
00027          DEV_DelayMs(2000);
00028
00029          (void)MDS_TaskDelay(150);
00030      }
00031  }

00038  void TEST_TestTask2(void)
00039  {
00040      while(1)
00041      {
00042          DEV_PutStrToMem((U8*)"\r\nTask2 is running! Tick is: %d",
00043                          MDS_SystemTickGet());
00044
00045          DEV_DelayMs(1000);
00046
00047          (void)MDS_TaskDelay(100);
00048      }
00049  }
```

```
00056  void TEST_TestTask3(void)
00057  {
00058      DEV_PutStrToMem((U8*)"₩r₩nTask3 is running! Tick is: %d", MDS_SystemTickGet());
00059
00060      /* 唤醒 task4 */
00061      (void)MDS_TaskWake(gpstrTask4Tcb);
00062
00063      while(1)
00064      {
00065          DEV_PutStrToMem((U8*)"\r\nTask3 is running! Tick is: %d",
00066                          MDS_SystemTickGet());
00067
00068          DEV_DelayMs(5000);
00069
00070          (void)MDS_TaskDelay(500);
00071      }
00072  }
```

```
00079  void TEST_TestTask4(void)
00080  {
00081      while(1)
00082      {
00083          DEV_PutStrToMem((U8*)"\r\nTask4 is running! Tick is: %d",
00084                          MDS_SystemTickGet());
00085
00086          DEV_DelayMs(1000);
00087
00088          (void)MDS_TaskDelay(1000);
00089      }
00090  }
```

不同的是 TEST_TestTask1 任务优先级为 4，创建时不使用任务选项参数，默认为 ready 态。TEST_TestTask2 任务优先级为 6，创建时使用任务选项参数，为 ready 态。TEST_TestTask3 任务优先级为 2，创建时使用任务选项参数，为 delay 态，延迟 2000 个 ticks，delay 态结束后会使用 MDS_TaskWake 函数唤醒 TEST_TestTask4 任务。TEST_TestTask4 任务优先级为 1，创建时使用任务选项参数，为 delay 态，永久延迟。

```
00014  void MDS_RootTask(void)
00015  {
00016      M_TASKOPT strOption;
00017
00018      /* 初始化软件 */
00019      DEV_SoftwareInit();
00020
00021      /* 初始化硬件 */
00022      DEV_HardwareInit();
00023
00024      /* 不使用 option 参数创建任务 1 */
00025      (void)MDS_TaskCreate((VFUNC)TEST_TestTask1, gaucTask1Stack, TASKSTACK, 4,
00026                           (M_TASKOPT*)NULL);
00027
00028      /* 使用 option 参数创建 ready 状态的任务 2 */
00029      strOption.ucTaskSta = TASKREADY;
00030      (void)MDS_TaskCreate((VFUNC)TEST_TestTask2, gaucTask2Stack, TASKSTACK, 6,
00031                           &strOption);
00032
00033      /* 使用 option 参数创建延迟 20 秒的 delay 状态的任务 3 */
00034      strOption.ucTaskSta = TASKDELAY;
```

```
00035       strOption.uiDelayTick = 2000;
00036       (void)MDS_TaskCreate((VFUNC)TEST_TestTask3, gaucTask3Stack, TASKSTACK, 2,
00037                            &strOption);
00038
00039       /* 使用 option 参数创建无限延迟的 delay 状态的任务 4 */
00040       strOption.ucTaskSta = TASKDELAY;
00041       strOption.uiDelayTick = DELAYWAITFEV;
00042       gpstrTask4Tcb = MDS_TaskCreate((VFUNC)TEST_TestTask4, gaucTask4Stack, TASKSTACK,
00043                                      1, &strOption);
00044
00045       (void)MDS_TaskDelay(DELAYWAITFEV);
00046   }
```

如果单从任务优先级来看，应该是 TEST_TestTask4 任务最先运行，但 TEST_TestTask4 任务处于永久 delay 状态，不能运行。剩下的优先级最高的任务是 TEST_TestTask3，但 TEST_TestTask3 任务也处于 delay 状态，需要延迟 2000 个 ticks 后才能重新参与调度。TEST_TestTask1 和 TEST_TestTask2 任务尽管使用了不同的任务参数，但都处于 ready 态，TEST_TestTask1 的优先级高，先运行 TEST_TestTask1，在 TEST_TestTask1 任务处于 delay 态时就会运行 TEST_TestTask2 任务。在前 2000 个 ticks，TEST_TestTask1 和 TEST_TestTask2 任务会交替运行，在 2000 个 ticks 时，TEST_TestTask3 任务 delay 时间耗尽，参与调度，应该可以看到 TEST_TestTask3 任务会打断其它任务开始运行。在 TEST_TestTask3 运行之后它立刻唤醒了 TEST_TestTask4 任务，应该还可以看到在 TEST_TestTask3 任务运行后立刻被 TEST_TestTask4 任务打断了，此后这 4 个任务按照优先级的调度方式交替运行。

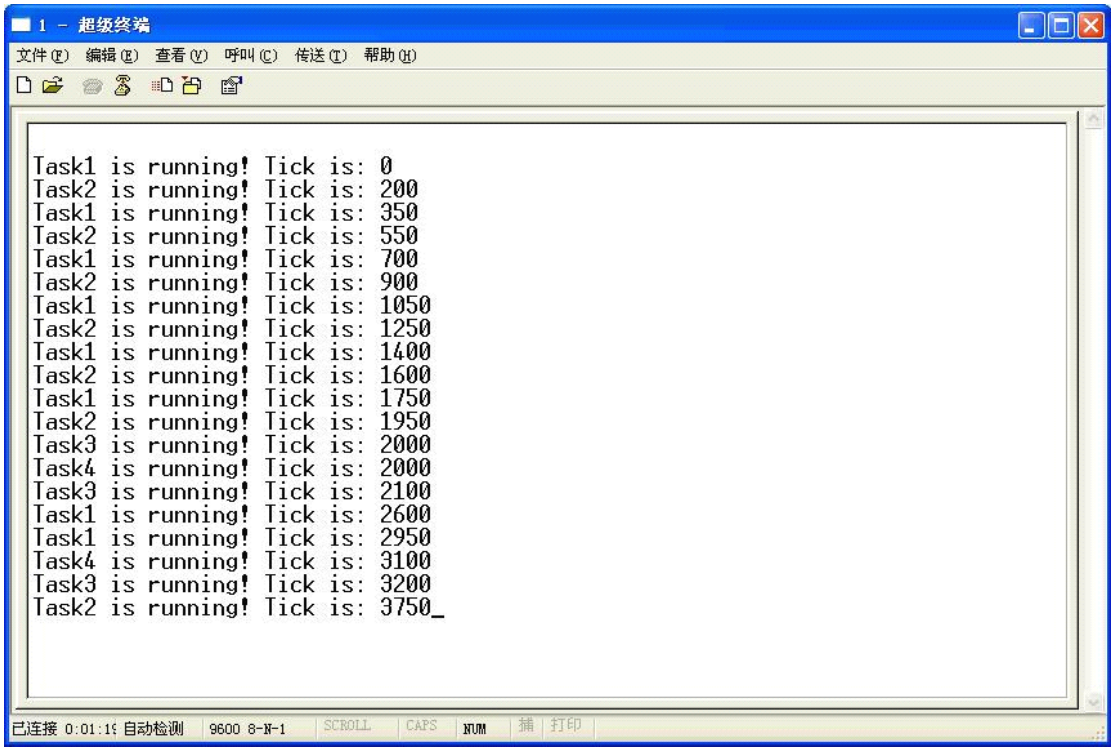


图 45　增加 delay 状态的 4 个任务运行结果

从图 45 中我们可以看到输出结果与我们设想的是一样的，TEST_TestTask1 任务最先开始运行，经过 200 个 ticks 进入 delay 状态，TEST_TestTask2 任务开始运行，此后 TEST_TestTask1 与 TEST_TestTask2 任务交替运行。在 2000 个 ticks 的时候 TEST_TestTask3 的 delay 时间耗尽，抢占其它任务开始运行，然后又被 TEST_TestTask4 任务抢占，之后 4

个任务参与任务调度。

读者可以从网站下载视频观看串口输出的过程。在视频中我们看到要么是在一段时间内没有字符打印出来，要么是几个任务的字符一起打印出来。这是因为打印字符的任务处于最低优先级，只有当其它任务都处于 delay 状态时它才可以运行，将内存中的字符打印到串口上来。

我们在这 4 个函数里使用了 DEV_DelayMs 函数模拟任务的业务功能，该函数会连续运行几秒时间，这段时间内的 CPU 占有率是 100%，因此字符打印到串口的时刻相比任务将字符打印到内存的时刻有较大的延迟，如果 CPU 占有率较低的话，我们可以看到几乎是实时的打印数据。

第 4 节 任务切换钩子函数

上节，我们引入了任务的 delay 态，通过最后例子的打印可以看到任务在交替运行，但这个打印只发生在每个任务每次循环的开始，看不到中间运行过程中任务的切换过程。

本小节将引入任务切换钩子函数，输出任务切换过程的打印信息。

钩子函数正如其名，它像钩子一样可以将函数挂在它上面，一旦它开始运行，它就会引发挂在它上面的函数也开始运行。按照这个思路，我们可以将一个钩子函数放到任务切换过程中，当我们需要输出任务切换过程的打印时，只需要将相关的打印函数挂到这个钩子函数上就可以了。

钩子函数的功能是使用一个指向函数的指针型全局变量实现的，在使用钩子函数前，需要使用钩子添加函数将需要执行的函数挂接到这个全局变量上，这个全局变量此时等效于被挂接的函数，因此运行这个全局变量就相当于是运行被挂接的函数了，来看代码：

定义一个任务切换的钩子全局变量：

```
VFHSWT gvfTaskSwitchHook;
```

其中 VFHSWT 是被挂接函数的类型：

```
typedef void    (*VFHSWT)(M_TCB*, M_TCB*);
```

被挂接的函数原型为：

```
void TEST_TaskSwitchPrint(M_TCB* pstrOldTcb, M_TCB* pstrNewTcb);
```

可以看到定义的全局变量 gvfTaskSwitchHook 与函数 TEST_TaskSwitchPrint 类型是完全一致的，它们都是同一种类型的函数指针。

在使用钩子函数前需要使用钩子初始化函数 MDS_TaskHookInit 将全局变量 gvfTaskSwitchHook 初始化为 NULL，以表明钩子变量没有挂接函数，不能运行。

```
00340  void MDS_TaskHookInit(void)
00341  {
00342      /* 初始化钩子变量 */
00343      gvfTaskSwitchHook = (VFHSWT)NULL;
00344  }
```

添加钩子函数时，将被添加函数 TEST_TaskSwitchPrint 作为参数传递给钩子添加函数

MDS_TaskSwitchHookAdd，MDS_TaskSwitchHookAdd(TEST_TaskSwitchPrint)，执行的过程就是将被添加函数的指针赋给全局变量，使全局变量等效于被添加的函数。

```
00351  void MDS_TaskSwitchHookAdd(VFHSWT vfFuncPointer)
00352  {
00353      gvfTaskSwitchHook = vfFuncPointer;
00354  }
```

删除钩子函数时，就是将全局变量置为 NULL，撇清与被添加函数的关系。

```
00361  void MDS_TaskSwitchHookDel(void)
00362  {
00363      gvfTaskSwitchHook = (VFHSWT)NULL;
00364  }
```

使用钩子函数时，若全局变量 gvfTaskSwitchHook 不为 NULL，说明已经挂接了函数，执行 gvfTaskSwitchHook 全局变量就相当于执行了被挂接的函数。

下面在任务调度函数 MDS_TaskSched 中增加了任务切换钩子功能，将打印任务切换的函数挂接到此钩子上即可打印出任务切换过程。

```
00352  void MDS_TaskSched(void)
00353  {
00354      M_TCB* pstrTcb;
00355
00356      /* 调度 delay 表任务 */
00357      MDS_TaskDelayTabSched();
00358
00359      /* 调度 ready 表任务 */
00360      pstrTcb = MDS_TaskReadyTabSched();
00361
00362  #ifdef MDS_INCLUDETASKHOOK
00363
00364      /* 如果任务切换钩子已经挂接函数则执行该函数 */
00365      if((VFHSWT)NULL != gvfTaskSwitchHook)
00366      {
00367          /* 不同任务之间切换才执行任务切换钩子函数 */
00368          if(pstrTcb != gpstrCurTcb)
00369          {
00370              gvfTaskSwitchHook(gpstrCurTcb, pstrTcb);
00371          }
00372      }
00373
00374  #endif
00375
00376      /* 准备任务切换 */
00377      MDS_TaskSwitch(pstrTcb);
00378  }
```

000362 行，若定义了 MDS_INCLUDETASKHOOK 宏，才执行任务切换钩子函数。其它与任务切换钩子相关的代码也被该宏包含了，若不想使用钩子功能，不定义该宏即可。

00365 行，若已添加了钩子函数，则走下面分支。

00368 行，若切换前的任务与切换后的任务是同一个任务，则不执行任务切换钩子函数，只有在发生不同任务切换时才调用任务切换钩子函数。

00370 行，执行钩子函数。

除了 MDS_TaskSched 函数，在 MDS_TaskReadySched 函数里也执行了任务调度功能，也需要增加 362~374 行的内容。

任务切换钩子函数 gvfTaskSwitchHook 在任务调度函数调度完成后运行，但它还是处于调度的中断中，因此不能将耗时长的函数挂接到任务切换钩子函数上。

钩子函数可以在代码运行时动态挂接，不需要更改代码，而且，如果需要更改钩子函数执行的功能时，只需要更改挂接到钩子上的函数就可以实现，非常方便。

钩子变量可以有不同的指针类型，但需要与被挂接的函数指针类型保持一致。

被挂接函数的打印信息包括了切换时刻，包括了从哪个任务切换到了哪个任务的信息：

```
00072  void TEST_TaskSwitchPrint(M_TCB* pstrOldTcb, M_TCB* pstrNewTcb)
00073  {
00074      DEV_PutStrToMem((U8*)"\r\nTask %s ---> Task %s! Tick is: %d",
00075                      pstrOldTcb->pucTaskName, pstrNewTcb->pucTaskName,
00076                      MDS_SystemTickGet());
00077  }
```

上面所使用的 pucTaskName 结构是任务名指针，它里面存放的是任务名字符串的指针。为了区分每个任务，本小节在 TCB 里增加了任务名指针 pucTaskName 这个结构。

```
typedef struct m_tcb
{
    STACKREG strStackReg;           /* 备份寄存器组 */
    M_TCBQUE strTcbQue;             /* TCB 结构队列 */
    M_TCBQUE strDelayQue;           /* delay 表队列 */
    U8* pucTaskName;                /* 任务名称指针 */
    U32 uiTaskFlag;                 /* 任务标志 */
    U8 ucTaskPrio;                  /* 任务优先级 */
    M_TASKOPT strTaskOpt;           /* 任务参数 */
    U32 uiStillTick;                /* 延迟到的时间 */
}M_TCB;
```

在创建任务时，保存有任务名的字符串的指针作为一个入口参数被传递给 MDS_TaskCreate 函数，该指针会被赋给 TCB 中的 pucTaskName 变量。如果创建的任务没有名称，则任务名参数需要将 NULL 传递给 MDS_TaskCreate 函数。

MDS_TaskCreate 函数的原型如下，代码改动较小，不再详细介绍。

```
M_TCB* MDS_TaskCreate(U8* pucTaskName, VFUNC vfFuncPointer, U8* pucTaskStack,
                      U32 uiStackSize, U8 ucTaskPrio, M_TASKOPT* pstrTaskOpt);
```

下面我们来看看验证本节功能的测试函数，测试函数 TEST_TestTask1~TEST_TestTask3，循环执行打印、运行、延迟这 3 个过程，它们会在运行过程中不断的发生切换，本节新增加的钩子函数会将这些切换过程打印出来。

```
00017  void TEST_TestTask1(void)
00018  {
00019      while(1)
00020      {
00021          DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00022                          MDS_SystemTickGet());
00023
00024          DEV_DelayMs(2000);
00025
```

```
00026          (void)MDS_TaskDelay(150);
00027      }
00028  }

00035  void TEST_TestTask2(void)
00036  {
00037      while(1)
00038      {
00039          DEV_PutStrToMem((U8*)"\r\nTask2 is running! Tick is: %d",
00040                          MDS_SystemTickGet());
00041
00042          DEV_DelayMs(1000);
00043
00044          (void)MDS_TaskDelay(100);
00045      }
00046  }

00053  void TEST_TestTask3(void)
00054  {
00055      while(1)
00056      {
00057          DEV_PutStrToMem((U8*)"\r\nTask3 is running! Tick is: %d",
00058                          MDS_SystemTickGet());
00059
00060          DEV_DelayMs(5000);
00061
00062          (void)MDS_TaskDelay(500);
00063      }
00064  }
```

TEST_TestTask1 函数不使用任务选项参数，默认为 ready 态。TEST_TestTask2 函数使用 ready 态的任务选项参数。TEST_TestTask3 函数使用任务选项参数，需要先 delay 2000 个 ticks。

```
00014  void MDS_RootTask(void)
00015  {
00016      M_TASKOPT strOption;
00017
00018      /* 初始化软件 */
00019      DEV_SoftwareInit();
00020
00021      /* 初始化硬件 */
00022      DEV_HardwareInit();
00023
00024      /* 不使用 option 参数创建任务 1 */
00025      (void)MDS_TaskCreate((U8*)"Test1", (VFUNC)TEST_TestTask1, gaucTask1Stack,
00026                           TASKSTACK, 2, (M_TASKOPT*)NULL);
00027
00028      /* 使用 option 参数创建 ready 状态的任务 2 */
00029      strOption.ucTaskSta = TASKREADY;
00030      (void)MDS_TaskCreate((U8*)"Test2", (VFUNC)TEST_TestTask2, gaucTask2Stack,
00031                           TASKSTACK, 3, &strOption);
00032
00033      /* 使用 option 参数创建延迟 20 秒的 delay 状态的任务 3 */
00034      strOption.ucTaskSta = TASKDELAY;
00035      strOption.uiDelayTick = 2000;
00036      (void)MDS_TaskCreate((U8*)"Test3", (VFUNC)TEST_TestTask3, gaucTask3Stack,
00037                           TASKSTACK, 1, &strOption);
```

```
00038
00039       (void)MDS_TaskDelay(10000);
00040
00041   #ifdef MDS_INCLUDETASKHOOK
00042
00043       /* 系统运行 100 秒后删除任务切换钩子函数 */
00044       MDS_TaskSwitchHookDel();
00045
00046   #endif
00047
00048       (void)MDS_TaskDelay(DELAYWAITFEV);
00049   }
```

00019 行，DEV_SoftwareInit 函数里会使用钩子添加函数 MDS_TaskSwitchHookAdd 将打印切换过程的 TEST_TaskSwitchPrint 函数添加到钩子变量 gvfTaskSwitchHook 上。

00039 行，root 任务 delay 10000 个 ticks，在这 1000 个 ticks 时间内，3 个测试任务将不断的发生任务切换，钩子函数会打印出这些切换过程。

00044 行，删除任务切换钩子函数，此后将不再打印任务切换过程。

root 任务具有最高的优先级，当 root 任务进入 delay 状态时，操作系统发生第一次任务切换，按照任务状态和任务优先级，我们应该会看到任务切换钩子函数打印出 root 任务切换到 TEST_TestTask1 任务的信息，然后 TEST_TestTask1 任务开始运行，输出打印信息。TEST_TestTask1 任务运行 200 个 ticks 后，调用 MDS_TaskDelay 函数进入 delay 状态，这时候应该是切换到 TEST_TestTask2 任务，我们应该可以看到 TEST_TestTask1 任务切换到 TEST_TestTask2 任务的信息。随后 TEST_TestTask2 任务开始运行，输出 TEST_TestTask2 任务的打印。此后这 2 个任务不断的交替运行，当这 2 个任务都处于 delay 状态时，idle 任务开始运行，将前面切换过程的信息从内存打印到串口上。

当系统运行到 2000 ticks 时，TEST_TestTask3 任务 delay 时间耗尽，开始运行，它比其它 2 个任务具有更高的优先级，会抢占正在运行的任务，我们应该可以看到任务切换到了 TEST_TestTask3，并输出了 TEST_TestTask3 的打印信息。此后这 3 个任务不断的交替运行。

当系统运行到 10000 ticks 时，任务切换钩子函数被删除，此后我们不会再看到任务切换过程的打印，只能看到任务每次循环的打印。

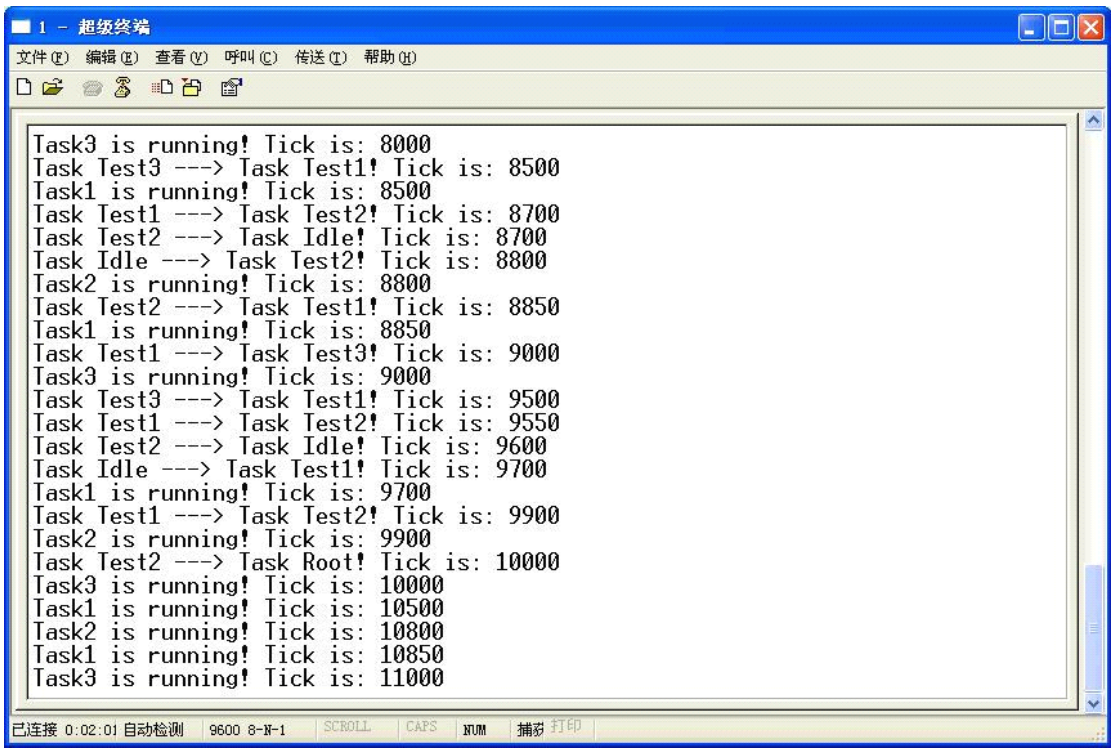


图 46　带有任务切换过程的打印

图 46 是一部分串口打印数据，读者可从网站下载视频，观看全部数据的打印过程，可以看到实际打印输出的结果与我们推断的结果是一致的。

我在串口工具中将本节的打印数据捕获了下来，保存到 4.4.txt 文件里，并使用 VC 编写了一个小工具，该工具可以解析 4.4.txt 文件里任务切换过程的数据，描绘出任务切换的过程，使不同任务之间的切换过程看的更直观些。

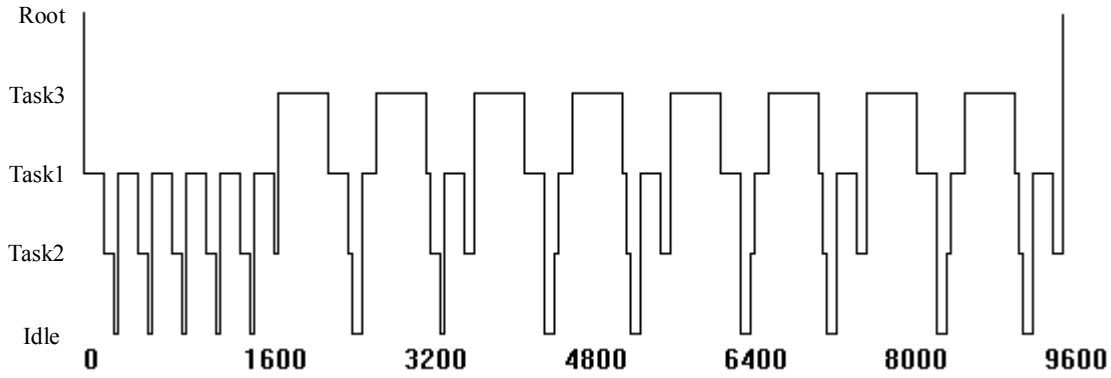


图 47　4.4 节任务切换过程图

第 5 节 任务创建和任务删除钩子函数

上节增加了任务切换钩子功能，打印出任务切换的信息，本节将增加任务创建钩子函数和任务删除钩子函数，分别打印出任务创建和删除的信息。

任务创建和删除钩子函数的原理与任务切换钩子函数的原理是一样的，都是需要定义一个与被挂接函数类型一致的指针型全局变量，使用钩子添加函数将被执行的函数挂接到钩子变量上，不同的是任务创建钩子函数在任务创建后打印，任务删除钩子函数在任务删除后打印。

钩子变量定义如下：

```
VFHCRT gvfTaskCreateHook;
VFHDLT gvfTaskDeleteHook;
```

VFHCRT 和 VFHDLT 的定义为：

```
typedef void    (*VFHCRT)(M_TCB*);
typedef void    (*VFHDLT)(M_TCB*);
```

被挂接的任务创建函数如下，与任务创建全局变量 gvfTaskCreateHook 类型一致，它会打印被创建函数的名称及创建的时间。

```
00071  void TEST_TaskCreatePrint(M_TCB* pstrTcb)
00072  {
00073      /* 创建任务成功 */
00074      if((M_TCB*)NULL != pstrTcb)
00075      {
00076          DEV_PutStrToMem((U8*)"\r\nTask %s is created! Tick is: %d",
00077                          pstrTcb->pucTaskName, MDS_SystemTickGet());
00078      }
00079      else /* 创建任务失败 */
00080      {
00081          DEV_PutStrToMem((U8*)"\r\nFail to create task! Tick is: %d",
00082                          MDS_SystemTickGet());
00083      }
00084  }
```

被挂接的任务删除函数如下，与任务删除全局变量 gvfTaskDeleteHook 类型一致，它会打印被删除函数的名称及删除的时间。

```
00104  void TEST_TaskDeletePrint(M_TCB* pstrTcb)
00105  {
00106      DEV_PutStrToMem((U8*)"\r\nTask %s is deleted! Tick is: %d",
00107                      pstrTcb->pucTaskName, MDS_SystemTickGet());
00108  }
```

如果需要使用钩子创建函数则需要在创建任务的函数 MDS_TaskCreate 里增加钩子函数代码，如下：

```
00023  M_TCB* MDS_TaskCreate(U8* pucTaskName, VFUNC vfFuncPointer, U8* pucTaskStack,
00024                        U32 uiStackSize, U8 ucTaskPrio, M_TASKOPT* pstrTaskOpt)
00025  {
00026
......       ......
00094
00095      /* 创建非系统任务后的操作 */
00096      if((MDS_RootTask != vfFuncPointer) && (MDS_IdleTask != vfFuncPointer))
00097      {
00098  #ifdef MDS_INCLUDETASKHOOK
00099
```

```
00100           /* 如果任务创建钩子已经挂接函数则执行该函数 */
00101           if((VFHCRT)NULL != gvfTaskCreateHook)
00102           {
00103               gvfTaskCreateHook(pstrTcb);
00104           }
00105
00106   #endif
00107
00108           /* 使用软中断调度任务 */
00109           MDS_TaskSwiSched();
00110       }
00111
00112       return pstrTcb;
00113   }
```

00026~00094 行，代码与上节完全一样。

00096 行，只有非系统任务才执行任务创建钩子函数。

00101~00104 行，若挂接了任务创建钩子函数则执行。

本节我们需要增加一个删除任务的函数 MDS_TaskDelete，用它来删除任务，在删除任务时可以执行挂接的任务删除钩子函数。

任务创建时在任务栈里创建了 TCB，TCB 中的 ready 节点和 delay 节点可能会挂接到 ready 表和 delay 表，我们只要将任务从 ready 表和 delay 表拆除，操作系统就无法再调度这个任务了，这样就可以删除该任务。删除后任务的任务栈中的数据还是存在的，但由于任务已经无法与 ready 表和 delay 表关联，因此这些数据就变成了无效数据。任务删除时，任务所使用的任务栈内存应该也被释放掉，但目前任务栈的内存是由用户申请的而不是由操作系统申请的，因此需要用户自行释放任务栈内存，操作系统不进行管理。以后任务栈由操作系统申请时，MDS_TaskDelete 函数才负责释放这部分内存。

```
00121   U32 MDS_TaskDelete(M_TCB* pstrTcb)
00122   {
00123       M_CHAIN* pstrChain;
00124       M_CHAIN* pstrNode;
00125       M_PRIOFLAG* pstrPrioFlag;
00126       U8 ucTaskPrio;
00127       U8 ucTaskSta;
00128
00129       /* 入口参数检查 */
00130       if(NULL == pstrTcb)
00131       {
00132           return RTN_FAIL;
00133       }
00134
00135       /* idle 任务不能被删除 */
00136       if(pstrTcb == gpstrIdleTaskTcb)
00137       {
00138           return RTN_FAIL;
00139       }
00140
00141       (void)MDS_IntLock();
00142
00143       /* 获取要删除任务的任务状态 */
00144       ucTaskSta = pstrTcb->strTaskOpt.ucTaskSta;
00145
00146       /* 该任务在 ready 表中，从 ready 表删除 */
```

```
00147       if(TASKREADY == (TASKREADY & ucTaskSta))
00148       {
00149           /* 获取该任务的相关调度参数 */
00150           ucTaskPrio = pstrTcb->ucTaskPrio;
00151           pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00152           pstrPrioFlag = &gstrReadyTab.strFlag;
00153
00154           /* 将该任务从 ready 表拆除 */
00155           (void)MDS_TaskDelFromSchedTab(pstrChain, pstrPrioFlag, ucTaskPrio);
00156       }
00157
00158       /* 任务在 delay 表则从 delay 表拆除 */
00159       if(DELAYQUEFLAG == (pstrTcb->uiTaskFlag & DELAYQUEFLAG))
00160       {
00161           /* 获取该任务 TCB 中挂接在 delay 调度表上的节点 */
00162           pstrNode = &pstrTcb->strDelayQue.strQueHead;
00163
00164           /* 从 delay 表拆除该任务 */
00165           (void)MDS_ChainCurNodeDelete(&gstrDelayTab, pstrNode);
00166       }
00167
00168       (void)MDS_IntUnlock();
00169
00170   #ifdef MDS_INCLUDETASKHOOK
00171
00172       /* 如果任务删除钩子已经挂接函数则执行该函数 */
00173       if((VFHDLT)NULL != gvfTaskDeleteHook)
00174       {
00175           gvfTaskDeleteHook(pstrTcb);
00176       }
00177
00178   #endif
00179
00180       /* 使用软中断调度任务 */
00181       MDS_TaskSwiSched();
00182
00183       return RTN_SUCD;
00184   }
```

00121 行，函数有 2 种返回值，RTN_SUCD 代表任务删除成功，RTN_FAIL 代表任务删除失败。入口参数 pstrTcb 是需要被删除的任务的 TCB 指针。

00130~00133 行，入口参数检查，入口参数为空则返回失败。

00136~00139 行，入口参数检查，若删除 idle 任务则返回失败。

00141 行，锁中断。下面将操作 ready 表和 delay 表，为防止多个任务同时操作这两个表，需要锁中断。

00144 行，获取被删除任务的当前任务状态。

00147~00156 行，若该任务在 ready 表中则从 ready 表删除。

00159~00166 行，若该任务在 delay 表中则从 delay 表删除。

00168 行，调度表操作完成，解锁中断。

00170~00178 行，若挂接了任务删除钩子函数则执行该函数。

00181 行，任务已经被删除，需要重新调度一次任务。

本节内容较少，主要内容已经介绍完毕。本节仍使用上节的 3 个功能验证函数，与上节不同的是在 8000 ticks 的时候 TEST_TestTask3 任务会被 root 任务删除，在 10000 ticks 时 root

任务也会被自己删除。

```
00014  void MDS_RootTask(void)
00015  {
00016      M_TCB* pstrTcb;
00017      M_TASKOPT strOption;
00018
00019      /* 初始化软件 */
00020      DEV_SoftwareInit();
00021
00022      /* 初始化硬件 */
00023      DEV_HardwareInit();
00024
00025      /* 不使用 option 参数创建任务 1 */
00026      (void)MDS_TaskCreate((U8*)"Test1", (VFUNC)TEST_TestTask1, gaucTask1Stack,
00027                           TASKSTACK, 2, (M_TASKOPT*)NULL);
00028
00029      /* 使用 option 参数创建 ready 状态的任务 2 */
00030      strOption.ucTaskSta = TASKREADY;
00031      (void)MDS_TaskCreate((U8*)"Test2", (VFUNC)TEST_TestTask2, gaucTask2Stack,
00032                           TASKSTACK, 3, &strOption);
00033
00034      /* 使用 option 参数创建延迟 20 秒的 delay 状态的任务 3 */
00035      strOption.ucTaskSta = TASKDELAY;
00036      strOption.uiDelayTick = 2000;
00037      pstrTcb = MDS_TaskCreate((U8*)"Test3", (VFUNC)TEST_TestTask3, gaucTask3Stack,
00038                               TASKSTACK, 1, &strOption);
00039
00040      (void)MDS_TaskDelay(8000);
00041
00042      /* 系统运行 80 秒后删除 task3 任务 */
00043      (void)MDS_TaskDelete(pstrTcb);
00044
00045      (void)MDS_TaskDelay(2000);
00046
00047      /* 系统运行 100 秒后删除 root 任务 */
00048      (void)MDS_TaskDelete(MDS_CurrentTcbGet());
00049  }
```

root 任务起来后创建了 3 个任务，我们会看到这 3 个任务创建过程的打印，随后任务运行的过程与上节一致。当系统运行到 8000 ticks 时，TEST_TestTask3 任务会被删除，我们可以看到 TEST_TestTask3 任务被删除的打印信息，此后就只有 TEST_TestTask1 和 TEST_TestTask2 任务在运行了。到 10000 ticks 时，root 任务调用 MDS_TaskDelete 函数将自己删除，无需再调用 MDS_TaskDelay(DELAYWAITFEV)函数进入永久 delay 状态。

本节运行结果截图如下：

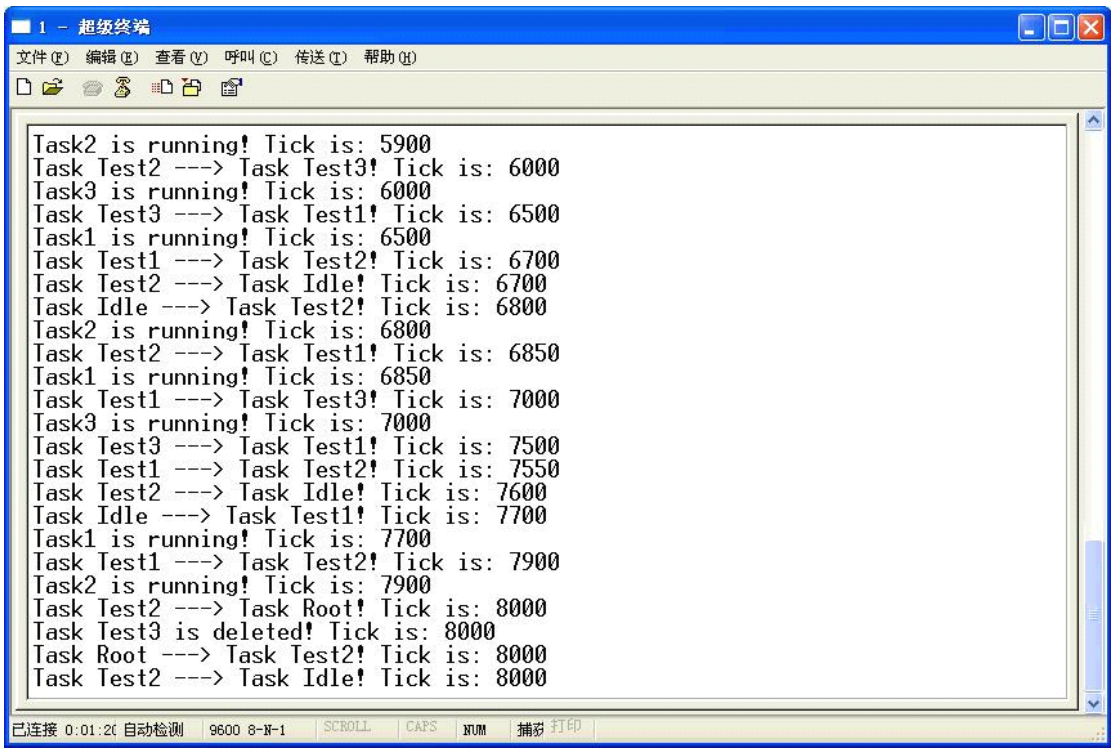


图 48　带有任务创建、切换和删除过程的打印

读者可以访问 http://blog.sina.com.cn/ifreecoding 网站下载视频，观看全部数据的打印过程，可以看到实际运行的结果与我们上面推断的结果是一致的。

使用工具软件将任务切换过程的数据解析为图形，结果如下图所示：

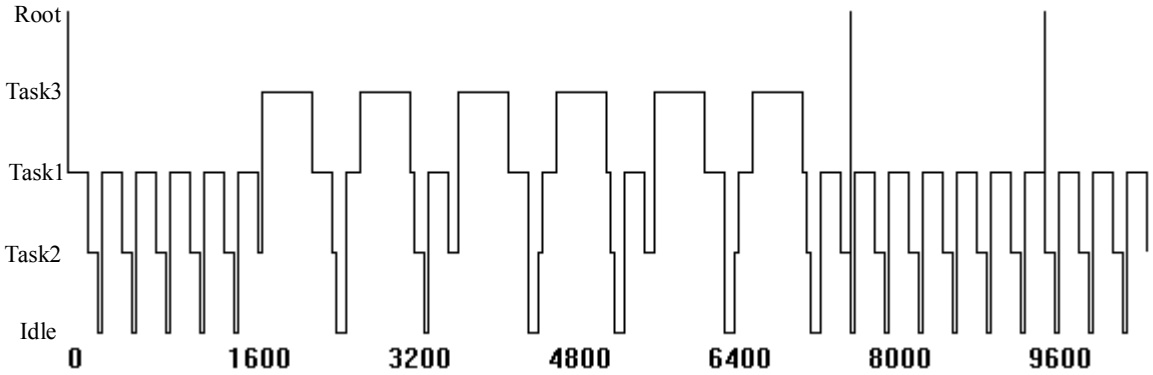


图 49　4.5 节任务切换过程图

从图中可以清晰的看到 2000 ticks 时 TEST_TestTask3 任务结束 delay 状态，开始运行，在 8000 ticks 时，root 任务 delay 时间耗尽，操作系统切换到 root 任务，删除了 TEST_TestTask3 任务，在 10000 ticks 时，root 任务 delay 时间再次耗尽，操作系统切换到 root 任务，删除了自己。此后只剩下 TEST_TestTask1 和 TEST_TestTask2 任务切换运行。

第 6 节 任务自结束

上节增加了删除任务的函数 MDS_TaskDelete，任务可以调用该函数结束其它任务或自身任务的运行。在前面章节我们说过，目前任务还不具备自结束功能，需要使用类似 while

的结构循环运行。本节我们将增加任务自结束功能，在创建任务时不再受任何限制。

任务要做到自结束，需要解决 2 个问题，一，任务需要能脱离操作系统的调度。二、任务在结束后，操作系统需要能发生调度，切换到下个任务继续运行。

这 2 个问题我们都可以像上节那样，调用 MDS_TaskDelete 函数解决，但如果使用 MDS_TaskDelete 函数来实现任务的自结束，那么 MDS_TaskDelete 函数就必须以显式的方式写在每个任务主函数的最后一行，当任务运行到结束时，任务会运行 MDS_TaskDelete 函数结束自身的运行。这样做有一个明显的缺点，我们需要为每个任务增加这样一行代码，不但对用户进行了约束，使用不方便，而且如果用户忘记了增加这行代码，那么操作系统运行到任务结束后就会崩溃掉。

前面我们介绍过，LR 寄存器中保存的是本函数返回上级父函数的 PC 指针，返回父函数时只要跳转到该 PC 指针就可以了。如果我们要做到任务自删除，那么就可以借用这个功能，可以将 MDS_TaskDelete 函数认为是每个创建任务所使用的主函数的父函数，在任务初始化时，将 MDS_TaskDelete 函数的地址赋给 LR 寄存器，这样当任务运行完最后一条指令时，它就会取出 LR 寄存器中的 MDS_TaskDelete 函数的地址，跳转到 MDS_TaskDelete 函数，实现任务的自结束功能。这种方式是隐式的，不会对用户做任何限制，只需要我们修改 MDS_TaskStackInit 函数代码，增加对 LR 寄存器的初始化就可以一劳永逸了。

```
00011  void MDS_TaskStackInit(M_TCB* pstrTcb, VFUNC vfFuncPointer)
00012  {
00013
……   ……
00032
00033      pstrRegSp->uiR14 = (U32)MDS_TaskSelfDelete; /* R14 */
00034
……   ……
00043
00044  }
```

MDS_TaskStackInit 函数的改动量非常小，只需要在 33 行将 MDS_TaskSelfDelete 函数赋给 LR 寄存器。MDS_TaskSelfDelete 函数封装了 MDS_TaskDelete 函数，将存放当前指针的全局变量 gpstrCurTcb 传给 MDS_TaskDelete 函数，完成任务的自删除。

```
00191  void MDS_TaskSelfDelete(void)
00192  {
00193      (void)MDS_TaskDelete(gpstrCurTcb);
00194  }
```

本节的改动已经介绍完毕，本节的验证函数与上节非常相似，其中 TEST_TestTask1 和 TEST_TestTask2 函数与上节完全一样，TEST_TestTask3 函数修改为只循环 6 次，每次循环时间为 1000 ticks。由于 TEST_TestTask3 函数具有最高优先级，并且在开始时 delay 了 2000 ticks，因此它会在 8000 ticks 时运行完毕，按照本节的设计，在 8000 ticks 时它应该会自结束运行，由任务删除钩子函数打印出任务删除的信息。

```
00053  void TEST_TestTask3(void)
00054  {
00055      U8 i;
00056
00057      /* Task3 任务运行 6 个循环后自动结束 */
00058      for(i = 0; i < 6; i++)
```

```
00059      {
00060          DEV_PutStrToMem((U8*)"\r\nTask3 is running! Tick is: %d",
00061                          MDS_SystemTickGet());
00062
00063          DEV_DelayMs(5000);
00064
00065          (void)MDS_TaskDelay(500);
00066      }
00067  }
```

除此之外 root 任务也不需要再调用 MDS_TaskDelete 函数自我删除，也是由本节新增加的隐式删除方式进行自删除。

```
00014  void MDS_RootTask(void)
00015  {
00016      M_TASKOPT strOption;
00017
00018      /* 初始化软件 */
00019      DEV_SoftwareInit();
00020
00021      /* 初始化硬件 */
00022      DEV_HardwareInit();
00023
00024      /* 不使用 option 参数创建任务 1 */
00025      (void)MDS_TaskCreate((U8*)"Test1", (VFUNC)TEST_TestTask1, gaucTask1Stack,
00026                           TASKSTACK, 2, (M_TASKOPT*)NULL);
00027
00028      /* 使用 option 参数创建 ready 状态的任务 2 */
00029      strOption.ucTaskSta = TASKREADY;
00030      (void)MDS_TaskCreate((U8*)"Test2", (VFUNC)TEST_TestTask2, gaucTask2Stack,
00031                           TASKSTACK, 3, &strOption);
00032
00033      /* 使用 option 参数创建延迟 20 秒的 delay 状态的任务 3 */
00034      strOption.ucTaskSta = TASKDELAY;
00035      strOption.uiDelayTick = 2000;
00036      (void)MDS_TaskCreate((U8*)"Test3", (VFUNC)TEST_TestTask3, gaucTask3Stack,
00037                           TASKSTACK, 1, &strOption);
00038
00039      (void)MDS_TaskDelay(10000);
00040
00041      /* 系统运行 100 秒后 root 任务自动结束 */
00042
00043  }
```

本节运行结果截图如下：

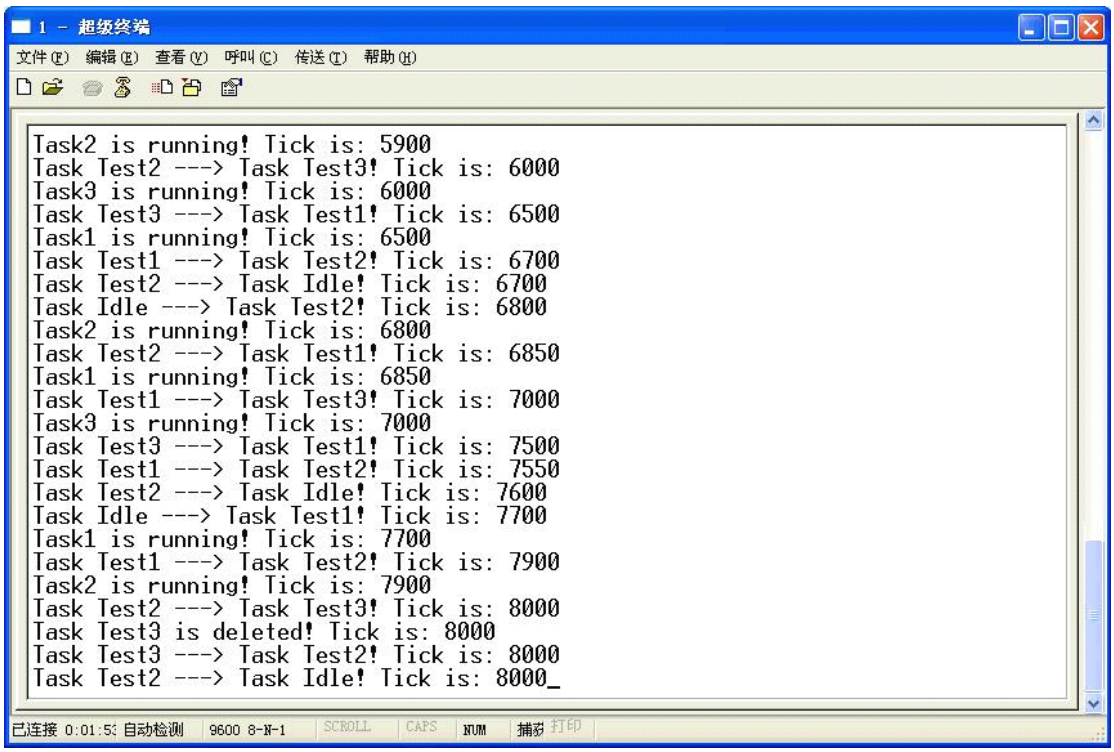


图 50　任务自删除的打印信息

读者可以访问 http://blog.sina.com.cn/ifreecoding 网站下载视频，观看全部数据的打印过程，可以看到本节实际运行打印出的结果与我们上节的实际运行打印出的结果是一致的，这也是与我们设计相符的，但这里还是有一点稍微的不同，见图 51：

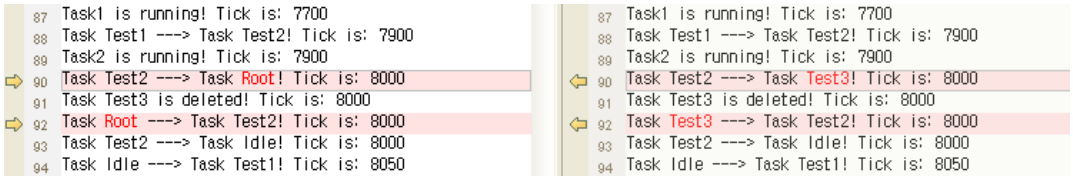


图 51　任务被删除与自删除的打印信息对比

图 51 左边是上节打印输出的数据，右边是本节打印输出的数据，这两组数据对比的结果如图 51 所示，只有 90 和 92 行的结果不同。上节当系统运行到 8000 ticks 时，root 任务 delay 时间耗尽，恢复到 running 状态，删除了 TEST_TestTask3 任务，之后又重新进入 delay 状态，将 CPU 控制权让给了 TEST_TestTask2 任务。而本节 TEST_TestTask3 任务先是 delay 了 2000 个 ticks，然后运行了 6 次 for 循环，6 次 for 循环后，系统时间为 8000 ticks，TEST_TestTask3 任务运行结束，自己结束了运行，将 CPU 控制权让给了 TEST_TestTask2 任务，因此本节删除 TEST_TestTask3 任务的操作是在 TEST_TestTask3 任务中完成的，只有这点与上节在 root 任务中删除 TEST_TestTask3 任务是不同的。

图 51 的截图是 Beyond Compare 工具软件的截图，Beyond Compare 是一款比较文件和文件夹的工具，它可以比较出不同源代码文件之间的细微差别并标记出来，即使你在成千上万行代码中只修改了 1 个字符，它也会发现并标记出来，这对于软件的维护有着非常重要的意义。

使用工具软件解析本节任务切换过程的数据，结果如下图所示：

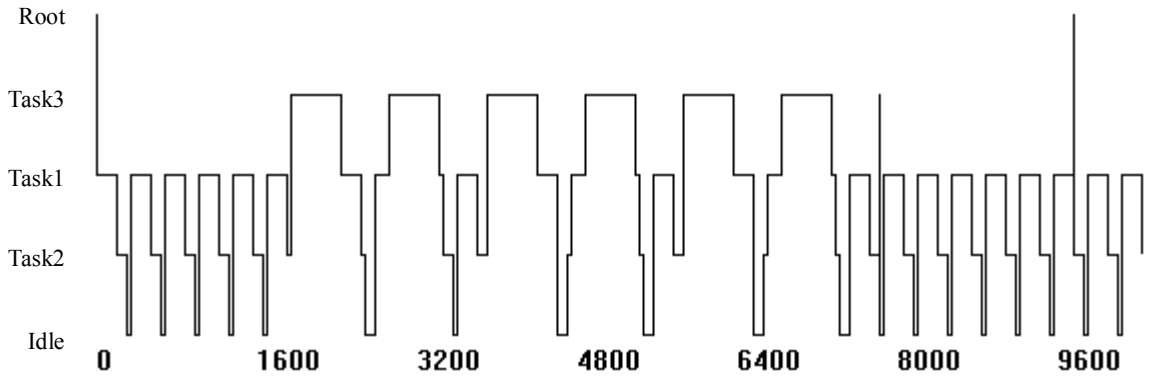


图 52　4.6 节任务切换过程图

对比上节中任务切换过程的图 49，可以明显的看出在 8000 ticks 时，本节是切换到了 TEST_TestTask3 任务，而上节是切换到了 root 任务。

第 7 节 二进制信号量

某些资源在同一时刻只可以被一个任务操作，实时操作系统的任务抢占特性会导致这些资源可能被多个任务同时操作，从而产生错误。本节将讲述二进制信号量的原理，可以利用二进制信号量保护这些资源，使多个任务只能串行的操作这些资源。

有时候我们可以设计一块共享内存，用来在多个任务间传递数据，比如使用任务 1 向共享内存中写入数据，使用任务 2 从这片内存中读取数据，这样就可以实现任务 1 向任务 2 传递数据的功能，但这样做有一个问题，如果任务 1 正在向共享内存中写数据的过程中发生了任务切换，切换到了任务 2，那么任务 2 所读取的共享内存中的数据就不完全是最新写入的有效数据，这样任务 2 就会读取到错误的数据。

为了防止这个问题发生，最简单的办法就是使用一个全局变量来指示共享内存的访问权限，当全局变量为 1 时，共享内存可以被访问，当全局变量为 0 时共享内存不可被访问。当一个任务操作共享内存时，首先判断全局变量，如果为 0，说明共享内存正在被其它任务操作，此时无法被访问，如果为 1 的话说明共享内存可以被访问，那么该任务则将全局变量置为 0，表明共享内存已经被访问，其它任务此时不可访问共享内存。当任务操作完共享内存后将全局变量置为 1，释放对共享内存的访问权限，此后共享内存又可以被再次访问。

这个过程使用伪码描述如下：

```
00001  锁中断;
00002
00003  /* 判断全局变量 */
00004  if(1 == 全局变量)
00005  {
00006      /* 获取到访问权限，将访问权限置为 0 */
00007      全局变量 = 0;
00008
00009      解锁中断;
00010  }
00011  else
00012  {
00013      解锁中断;
```

```
00014
00015       /* 无法获取到访问权限，函数返回 */
00016       return;
00017   }
00018
00019   对共享内存的操作;
00020
00021   /* 访问结束，将访问权限置为 1 */
00022   全局变量 = 1;
```

上述函数在运行时可能会发生重入现象，因此 4~7 行需要用锁中断的方式将全局变量的操作过程保护起来，虽然在 22 行也存在重入的问题，但 22 行只有一条指令涉及到对全局变量的操作，而不是一个过程，因此无需锁中断保护。这里所说的一条指令，不是指 C 语言的一条指令，而是汇编语言的一条指令。

二进制信号量就是基于上述原理实现的，简单来说，二进制信号量就是一个全局变量，用来实现各种资源的互斥，但使用全局变量作为资源互斥的开关存在一个缺点：当任务获取不到访问权限时，它可能需要等待该权限，需要暂时放弃 CPU 资源，让给其它任务去运行，这就需要发生任务调度，但直接触发任务调度的软中断调度函数被封装到了操作系统内部，用户不可见，因此获取不到权限的任务也就无法主动发生任务调度切换到其它任务。

二进制信号量将任务调度函数封装到了其内部，当任务获取不到权限被阻塞时可以直接调用软中断函数 MDS_TaskSwiSched 触发任务调度函数，切换到其它任务继续运行，因此，可以说二进制信号量就是全局变量+任务调度的结合体。

在这节，我们将引入任务的另一个状态，阻塞态（pend），当任务获取不到信号量资源时就会进入 pend 态，pend 态与 delay 态非常相似，delay 态是由任务主动释放 CPU 资源而进入的等待状态，而 pend 态则是由于任务获取不到某些非 CPU 资源而被动进入的等待状态。如果处于 pend 态的任务不是永久 pend 状态，那么该任务也将被挂入 delay 表中，与处于非永久 delay 状态的任务一起参与 tick 中断的调度。当 pend 的时间耗尽时，任务将会被从 delay 表拆除，结束 pend 状态，重新挂入 ready 表，参与任务调度。

在使用信号量前，需要使用 MDS_SemCreate 函数创建信号量，新创建的信号量可以为“空”状态或者“满”状态，为与后续章节增加的信号量保持兼容，我们将“空”定义为 0，将“满”定义为 0xFFFFFFFF，而不是 1。任务需要使用 MDS_SemTake 函数获取信号量，获取信号量的过程就是信号量从满到空的过程。任务需要使用 MDS_SemGive 函数释放信号量，释放信号量的过程就是信号量从空到满的过程。信号量空满状态对信号量操作的对应关系如表 9 所示：

操作方式	操作后状态	发生的结果
MDS_SemCreate	满	信号量被初始化为满状态。
MDS_SemTake	空	任务获取到信号量。
MDS_SemTake	空	任务没有获取到信号量，被阻塞。
MDS_SemTake	空	任务没有获取到信号量，共有 2 个任务被阻塞。
MDS_SemGive	空	一个任务获取到信号量，重新恢复到 ready 态，还有一个任务被阻塞。
MDS_SemGive	空	一个任务获取到信号量，重新恢复到 ready 态，没有任务被阻塞。
MDS_SemGive	满	没有任务获取信号量，信号量被置为满状态。
MDS_SemGive	满	没有任务获取信号量，信号量仍为满状态。

表 9　信号量操作与二进制信号量空满状态的对应关系

一个信号量可以阻塞多个任务，当信号量为空时，任何任务使用 MDS_SemTake 函数都可能被阻塞到该信号量上。当信号量上有被阻塞的任务时，如果一个任务使用 MDS_SemGive 函数释放了信号量，那么在这些被阻塞的任务中将会有一个任务被激活，从 pend 态恢复到 ready 态，重新参与任务调度。但具体是哪个任务先从 pend 状态恢复，我们可以采用两种调度方式，一种是先进先出（FIFO）方式，即最先被阻塞的任务最先被从 pend 状态恢复，另一种是优先级（PRIO）方式，即被阻塞的任务中优先级最高的任务最先被恢复。前面介绍的操作系统的任务调度方式就是优先级方式，因此在信号量里我们仍可以使用与任务调度相同的 ready 表结构来实现信号量的优先级调度，每个信号量里都有一个类似 ready 表的调度表，当任务被信号量阻塞时，任务被从 ready 表拆除，被挂接到信号量的调度表中，当信号量被释放时，激活信号量调度表中的最高优先级任务，将它从信号量表拆除，挂接到 ready 表中，对信号量调度表的拆除、添加过程与对任务调度表的拆除、添加过程是一样的。

信号量结构如下所示：

```
typedef struct m_sem
{
    M_TASKSCHEDTAB strSemTab;       /* 信号量调度表 */
    U32 uiCounter;                  /* 信号量计数值 */
    U32 uiSemOpt;                   /* 信号量参数 */
}M_SEM;
```

其中 M_TASKSCHEDTAB 结构就是 ready 表的结构，可以将被阻塞的任务按照任务调度的方式挂接到 strSemTab 变量上。uiCounter 变量用来表明信号量是空还是满。uiSemOpt 变量用来表明信号量是采用 FIFO 还是 PRIO 调度方式。

当信号量采用 FIFO 调度方式时，它只需要一个根节点就可以了，所有被阻塞的任务按照阻塞的顺序挂接到尾节点上，任务恢复时从头节点依次取出。在 FIFO 调度方式中，我们只使用 strSemTab 中优先级 0 的根节点作为 FIFO 方式的根节点就可以了。

在获取信号量时，如果暂时获取不到信号量，那么有的情况可能需要任务一直处于 pend 状态，一直等待获取到信号量后才重新返回 ready 状态重新参与任务调度。而有的情况则可能会需要设定一个超时上限，如果任务在超时时间内获取到信号量，那么任务会返回 ready 状态重新参与调度，如果超时时间耗尽时还没有获取到信号量，那么任务也会重新转换为 ready 态重新参与任务调度。而有的情况则可能不需要做任何时间等待，任务获取不到信号量的话也需要继续运行。

上面列出了二进制信号量所需要实现的所有功能，下面我们来看看代码的具体实现过程。

在使用信号量前需要先定义一个 M_SEM 型的信号量变量，使用信号量初始化函数 MDS_SemCreate 对该变量进行初始化：

```
00019  U32 MDS_SemCreate(M_SEM* pstrSem, U32 uiSemOpt, U32 uiInitVal)
00020  {
00021      /* 入口参数检查 */
00022      if(NULL == pstrSem)
00023      {
00024          return RTN_FAIL;
00025      }
00026
00027      /* 信号量选项检查 */
```

```
00028       if((SEMFIFO != (SEMSCHEDOPTMASK & uiSemOpt))
00029          && (SEMPRIO != (SEMSCHEDOPTMASK & uiSemOpt)))
00030       {
00031           return RTN_FAIL;
00032       }
00033
00034       /* 二进制信号量初始值只能是空或者满 */
00035       if((SEMEMPTY != uiInitVal) && (SEMFULL != uiInitVal))
00036       {
00037           return RTN_FAIL;
00038       }
00039
00040       /* 初始化信号量调度表 */
00041       MDS_TaskSchedTabInit(&pstrSem->strSemTab);
00042
00043       /* 初始化信号量初始值 */
00044       pstrSem->uiCounter = uiInitVal;
00045
00046       /* 初始化信号量参数 */
00047       pstrSem->uiSemOpt = uiSemOpt;
00048
00049       return RTN_SUCD;
00050   }
```

00019 行，函数返回值包括 RTN_SUCD，表明创建信号量成功，RTN_FAIL 表明创建信号量失败。入口参数 pstrSem 为需要初始化的信号量的指针，入口参数 uiSemOpt 为创建信号量所使用的选项，创建先进先出的信号量时使用 SEMFIFO 选项，创建优先级的信号量时使用 SEMPRIO 信选项。uiInitVal 是信号量的初始值，SEMEMPTY 表明创建的信号量的初始值为空，SEMFULL 表明创建的信号量的初始值为满。

00022~00025 行，对入口参数 pstrSem 进行检查，若为空则返回失败。

00028~00032 行，对入口参数 uiSemOpt 进行检查，若既不是 FIFO 状态也不是 PRIO 状态则返回失败。

00035~00038 行，对入口参数 uiInitVal 进行检查，若信号量初始化值既不是空也不是满则返回失败。

00041 行，初始化信号量中的调度表，这个过程与任务中初始化 ready 表的过程是一致的。

00044 行，初始化信号量的初始值。

00047 行，初始化信号量的参数。

获取信号量的函数 MDS_SemTake 的代码如下：

```
00069   U32 MDS_SemTake(M_SEM* pstrSem, U32 uiDelayTick)
00070   {
00071       /* 入口参数检查 */
00072       if(NULL == pstrSem)
00073       {
00074           return RTN_FAIL;
00075       }
00076
00077       (void)MDS_IntLock();
00078
00079       /* 更新与当前任务相关的信号量 */
00080       gpstrCurTcb->pstrSem = pstrSem;
00081
```

```
00082       /* 获取信号量时不等待时间 */
00083       if(SEMNOWAIT == uiDelayTick)
00084       {
00085           /* 信号量为满，可获取到信号量 */
00086           if(SEMFULL == pstrSem->uiCounter)
00087           {
00088               /* 获取到信号量后将信号量置为空 */
00089               pstrSem->uiCounter = SEMEMPTY;
00090
00091               (void)MDS_IntUnlock();
00092
00093               return RTN_SUCD;
00094           }
00095           else /* 信号量为空，无法获取到信号量 */
00096           {
00097               (void)MDS_IntUnlock();
00098
00099               return RTN_SMTKRT;
00100           }
00101       }
00102       else /* 获取信号量时需要等待时间 */
00103       {
00104           /* 信号量为满，可获取到信号量 */
00105           if(SEMFULL == pstrSem->uiCounter)
00106           {
00107               /* 获取到信号量后将信号量置为空 */
00108               pstrSem->uiCounter = SEMEMPTY;
00109
00110               (void)MDS_IntUnlock();
00111
00112               return RTN_SUCD;
00113           }
00114           else /* 信号量为空，无法获取到信号量，需要切换任务 */
00115           {
00116               /* 将任务置为 pend 状态 */
00117               if(RTN_FAIL == MDS_TaskPend(pstrSem, uiDelayTick))
00118               {
00119                   (void)MDS_IntUnlock();
00120
00121                   /* 任务 pend 失败 */
00122                   return RTN_FAIL;
00123               }
00124
00125               (void)MDS_IntUnlock();
00126
00127               /* 使用软中断调度任务 */
00128               MDS_TaskSwiSched();
00129
00130               /* 任务 pend 的返回值，该值在任务 pend 状态结束时被保存在 uiDelayTick 中 */
00131               return gpstrCurTcb->strTaskOpt.uiDelayTick;
00132           }
00133       }
00134   }
```

00069 行，函数有 5 种返回值，RTN_SUCD：在延迟时间内获取到信号量。RTN_FAIL：获取信号量失败。RTN_SMTKTO：信号量时间耗尽，超时返回。RTN_SMTKRT：任务延迟状态被中断，没有获取到信号量。RTN_SMTKDL：信号量被删除。入口参数 pstrSem 为需要操作的信号量的指针。入口参数 uiDelayTick 为获取不到信号量时的超时时间，超时时间

分为 SEMNOWAIT、SEMWAITFEV 和任意数值这 3 种类型，SEMNOWAIT 是不等待，若获取不到信号量则立刻执行下条指令，SEMWAITFEV 是永久等待，若获取不到信号量则永久处于 pend 状态，任意数值为 pend 的超时时间，单位为 tick，若在此时间内获取到信号量，则任务重新返回 ready 态参与任务调度，若超时时间耗尽了还没有获取信号量，那么任务也重新返回 ready 态参与任务调度，这两种情况的返回值不同，用户可以根据返回值做相应的处理。

00072~00075 行，对入口参数 pstrSem 进行检查，若为空则返回失败。

00077 行，锁中断，防止多个任务同时操作信号量。

00080 行，将阻塞任务的信号量赋给 TCB 中相关的变量。本节在 TCB 中新增加了一个 M_SEM*型的变量，

```
typedef struct m_tcb
{
    STACKREG strStackReg;           /* 备份寄存器组 */
    M_TCBQUE strTcbQue;             /* TCB 结构队列 */
    M_TCBQUE strDelayQue;           /* delay 表队列 */
    M_SEM* pstrSem;                 /* 与任务相关的信号量指针 */
    U8* pucTaskName;                /* 任务名称指针 */
    U32 uiTaskFlag;                 /* 任务标志 */
    U8 ucTaskPrio;                  /* 任务优先级 */
    M_TASKOPT strTaskOpt;           /* 任务参数 */
    U32 uiStillTick;                /* 延迟到的时间 */
}M_TCB;
```

用它记录阻塞任务的信号量，这样，我们就可以通过这个变量从 TCB 中找到阻塞任务的信号量，进而找到信号量的调度表，然后就可以对阻塞这个任务的信号量的调度表进行操作了。

00083 行，pend 时间为 0 走此分支。

00086 行，若信号量处于满状态走此分支。

00089 行，信号量为满状态，可获取到信号量，任务获取到信号量后，将信号量置为空状态。

00091 行，对信号量操作完毕，解锁中断。

00093 行，对信号量操作完毕，返回 RTN_SUCD。

00095 行，信号量为空走此分支。

00097 行，走此分支说明信号量为空无法获取，并且 pend 的时间为 SEMNOWAIT，说明任务不需要进入 pend 状态，因此不对信号量做任何处理，直接解锁中断，准备返回。

00099 行，返回 RTN_SMTKRT 值，表明任务没有获取到信号量，直接返回。

00102 行，被 pend 的任务需要等待时间走此分支。

00105 行，若信号量处于满状态走此分支。

00108 行，信号量为满状态，可获取到信号量，任务获取到信号量后，将信号量置为空状态。

00110 行，对信号量操作完毕，解锁中断。

00112 行，对信号量操作完毕，返回 RTN_SUCD。

00114 行，信号量为空走此分支。

00117 行，走此分支说明信号量为空无法获取，需要等待一定时间以获取信号量，调用 MDS_TaskPend 函数操作各种调度表，将当前任务阻塞到 pstrSem 信号量上，阻塞时间为 uiDelayTick。

00119 行，阻塞任务操作发生错误，在函数返回前先解锁中断。

00122 行，对信号量操作失败，返回 RTN_FAIL。

00125 行，任务已经被阻塞，函数返回前先解锁中断。

00128 行，各种调度表在 117 行 MDS_TaskPend 函数里已经更新完毕，此处需要调用软中断函数进行任务调度。

00131 行，走到此行，说明任务已经从 running 态切换为 pend 态，又从 pend 态切换回 running 态，该函数的返回值已经被存入到当前任务 TCB 的 strTaskOpt.uiDelayTick 变量中，返回函数的返回值。

MDS_TaskPend 函数与 MDS_TaskDelay 函数的处理过程非常相似，主要是将任务从 ready 表拆除，添加到 delay 表中，细节不再做介绍，请读者自行参考代码：

```
00439  U32 MDS_TaskPend(M_SEM* pstrSem, U32 uiDelayTick)
00440  {
00441      M_CHAIN* pstrChain;
00442      M_CHAIN* pstrNode;
00443      M_CHAIN* pstrDelayNode;
00444      M_TCBQUE* pstrTaskQue;
00445      M_PRIOFLAG* pstrPrioFlag;
00446      U8 ucTaskPrio;
00447
00448      /* idle 任务不能处于 delay 状态 */
00449      if(gpstrCurTcb == gpstrIdleTaskTcb)
00450      {
00451          return RTN_FAIL;
00452      }
00453
00454      /* 获取当前任务的相关调度参数 */
00455      ucTaskPrio = gpstrCurTcb->ucTaskPrio;
00456      pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00457      pstrPrioFlag = &gstrReadyTab.strFlag;
00458
00459      /* 将当前任务从 ready 表拆除 */
00460      pstrNode = MDS_TaskDelFromSchedTab(pstrChain, pstrPrioFlag, ucTaskPrio);
00461
00462      /* 清除任务的 ready 状态 */
00463      gpstrCurTcb->strTaskOpt.ucTaskSta &= ~((U8)TASKREADY);
00464
00465      /* 更新当前任务的延迟时间 */
00466      gpstrCurTcb->strTaskOpt.uiDelayTick = uiDelayTick;
00467
00468      /* 非永久等待任务才挂入 delay 表 */
00469      if(SEMWAITFEV != uiDelayTick)
00470      {
00471          gpstrCurTcb->uiStillTick = guiTick + uiDelayTick;
00472
00473          /* 获取当前任务 delay 表的任务节点 */
00474          pstrTaskQue = (M_TCBQUE*)pstrNode;
00475          pstrDelayNode = &pstrTaskQue->pstrTcb->strDelayQue.strQueHead;
00476
00477          /* 将当前任务加入到 delay 表 */
00478          MDS_TaskAddToDelayTab(pstrDelayNode);
00479
00480          /* 置任务在 delay 表标志 */
00481          gpstrCurTcb->uiTaskFlag |= DELAYQUEFLAG;
00482      }
00483
```

```
00484       /* 将该任务添加到信号量调度表中 */
00485       MDS_TaskAddToSemTab(gpstrCurTcb, pstrSem);
00486
00487       /* 增加任务的 pend 状态 */
00488       gpstrCurTcb->strTaskOpt.ucTaskSta |= TASKPEND;
00489
00490       return RTN_SUCD;
00491   }
```

MDS_TaskPend 函数中所使用的 MDS_TaskAddToSemTab 函数的功能是将任务添加到信号量调度表中，如果信号量采用优先级调度方式，则使用 MDS_TaskAddToSchedTab 函数添加，这个过程与将任务添加到 ready 表的过程是一样的。如果信号量采用先进先出调度方式，则将任务添加到链表的尾节点上。这个过程比较简单，不再详细介绍，代码如下：

```
00353   void MDS_TaskAddToSemTab(M_TCB* pstrTcb, M_SEM* pstrSem)
00354   {
00355       M_CHAIN* pstrChain;
00356       M_CHAIN* pstrNode;
00357       M_PRIOFLAG* pstrPrioFlag;
00358       U8 ucTaskPrio;
00359
00360       /* 如果信号量是采用优先级调度算法 */
00361       if(SEMPRIO == (SEMSCHEDOPTMASK & pstrSem->uiSemOpt))
00362       {
00363           /* 获取任务的相关参数 */
00364           ucTaskPrio = pstrTcb->ucTaskPrio;
00365           pstrChain = &pstrSem->strSemTab.astrChain[ucTaskPrio];
00366           pstrNode = &pstrTcb->strTcbQue.strQueHead;
00367           pstrPrioFlag = &pstrSem->strSemTab.strFlag;
00368
00369           /* 添加到 sem 调度表 */
00370           MDS_TaskAddToSchedTab(pstrChain, pstrNode, pstrPrioFlag, ucTaskPrio);
00371       }
00372       else /* 如果信号量采用先进先出调度算法 */
00373       {
00374           /* 获取任务的相关参数，使用 0 优先级链表作为先进现出链表 */
00375           pstrChain = &pstrSem->strSemTab.astrChain[LOWESTPRIO];
00376           pstrNode = &pstrTcb->strTcbQue.strQueHead;
00377
00378           /* 添加到 sem 调度表 */
00379           MDS_ChainNodeAdd(pstrChain, pstrNode);
00380       }
00381   }
```

释放信号量的函数 MDS_SemGive 的代码如下：

```
00143   U32 MDS_SemGive(M_SEM* pstrSem)
00144   {
00145       M_TCB* pstrTcb;
00146       M_CHAIN* pstrChain;
00147       M_CHAIN* pstrNode;
00148       M_PRIOFLAG* pstrPrioFlag;
00149       U8 ucTaskPrio;
00150
00151       /* 入口参数检查 */
00152       if(NULL == pstrSem)
```

```
00153     {
00154         return RTN_FAIL;
00155     }
00156 
00157     (void)MDS_IntLock();
00158 
00159     /* 信号量为空 */
00160     if(SEMEMPTY == pstrSem->uiCounter)
00161     {
00162         /* 在被该信号量阻塞的任务中获取需要释放的任务 */
00163         pstrTcb = MDS_SemGetActiveTask(pstrSem);
00164 
00165         /* 有阻塞的任务, 释放任务 */
00166         if(NULL != pstrTcb)
00167         {
00168             /* 从信号量调度表拆除该任务 */
00169             (void)MDS_TaskDelFromSemTab(pstrTcb);
00170 
00171             /* 任务在 delay 表则从 delay 表拆除 */
00172             if(DELAYQUEFLAG == (pstrTcb->uiTaskFlag & DELAYQUEFLAG))
00173             {
00174                 pstrNode = &pstrTcb->strDelayQue.strQueHead;
00175                 (void)MDS_ChainCurNodeDelete(&gstrDelayTab, pstrNode);
00176 
00177                 /* 置任务不在 delay 表标志 */
00178                 pstrTcb->uiTaskFlag &= (~((U32)DELAYQUEFLAG));
00179             }
00180 
00181             /* 清除任务的 pend 状态 */
00182             pstrTcb->strTaskOpt.ucTaskSta &= ~((U8)TASKPEND);
00183 
00184             /* 借用 uiDelayTick 变量保存 pend 任务的返回值 */
00185             pstrTcb->strTaskOpt.uiDelayTick = RTN_SUCD;
00186 
00187             /* 将该任务添加到 ready 表中 */
00188             pstrNode = &pstrTcb->strTcbQue.strQueHead;
00189             ucTaskPrio = pstrTcb->ucTaskPrio;
00190             pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00191             pstrPrioFlag = &gstrReadyTab.strFlag;
00192 
00193             MDS_TaskAddToSchedTab(pstrChain, pstrNode, pstrPrioFlag, ucTaskPrio);
00194 
00195             /* 增加任务的 ready 状态 */
00196             pstrTcb->strTaskOpt.ucTaskSta |= TASKREADY;
00197 
00198             (void)MDS_IntUnlock();
00199 
00200             /* 使用软中断调度任务 */
00201             MDS_TaskSwiSched();
00202 
00203             return RTN_SUCD;
00204         }
00205         else /* 没有阻塞的任务, 将信号量置为满 */
00206         {
00207             pstrSem->uiCounter = SEMFULL;
00208         }
00209     }
00210 
00211     (void)MDS_IntUnlock();
```

```
00212
00213       return RTN_SUCD;
00214   }
```

00143 行，函数返回值 RTN_SUCD 代表释放信号量成功，RTN_FAIL 代表释放信号量失败。入口参数 pstrSem 为释放的信号量的指针。

00152~00155 行，对入口参数 pstrSem 进行检查，若为空则返回失败。

00157 行，锁中断，防止多个任务同时操作信号量。

00160 行，若信号量处于空状态走此分支。

00163 行，从信号量 pstrSem 的调度表里面被阻塞的任务中查找需要被释放的任务，返回该任务的 TCB。

00166 行，若该任务的 TCB 不为空，说明有被阻塞的任务，走此分支。

00169 行，将这个任务从信号量调度表中删除。

00172 行，如果这个任务处于 delay 表中，走此分支。

00174 行，获取这个任务 TCB 中挂接到 delay 表的节点。

00175 行，将该任务从 delay 表中删除。

00178 行，将该任务的标志修改为不在 delay 表中。

00182 行，清除该任务的 pend 状态。

00185 行，将该任务的返回值保存在 TCB 的 strTaskOpt.uiDelayTick 中，当该任务重新运行时就会得到 RTN_SUCD 这个返回值。注意，该任务是当前运行任务使用信号量释放出来的任务，而不是当前正在运行的任务。

00188~00196 行，将该任务添加到 ready 表中，并将任务的状态修改为 ready 状态。

00198 行，对信号量操作完毕，解锁中断。

00201 行，已经有任务新加入到 ready 表中，使用软中断函数调度任务。

00205 行，信号量中没有被阻塞的任务，走此分支。

00207 行，走到此分支，说明信号量是在空的状态下被释放，并且没有被阻塞的任务需要获取信号量，因此直接将信号量置为满状态。

00211 行，对信号量操作完毕，解锁中断。

00213 行，对信号量操作完毕，返回释放信号量成功。

如果有任务被信号量阻塞，在中断中使用 MDS_SemGive 函数释放信号量时并不会立刻发生任务调度，因为 MDS_SemGive 函数所使用的软中断调度函数 MDS_TaskSwiSched 无法在中断中执行，任务调度需要等到下个 tick 中断到来时才能执行，因此在中断中释放信号量激活的最高优先级任务可能会有一个小的延迟后才能运行。在后面章节我们会将 Mindows 移植到 cortex 内核的芯片上，cortex 内核拥有 pendsv 软中断，使用 pendsv 软中断可以解决这个问题，不会有延迟产生。

MDS_SemGetActiveTask 函数的功能是从信号量的调度表中获取需要被最先释放的任务，如果信号量采用的是优先级调度方式，则使用 MDS_TaskHighestPrioGet 函数从信号量调度表中找出最高优先级任务的 TCB，如果信号量采用的是先进先出的调度方式，则从信号量调度表中取出最先入链表的头节点。函数比较简单，不再详细介绍，代码如下：

```
00424   M_TCB* MDS_SemGetActiveTask(M_SEM* pstrSem)
00425   {
00426       M_CHAIN* pstrNode;
00427       M_TCBQUE* pstrTaskQue;
00428       U8 ucTaskPrio;
```

```
00429
00430       /* 信号量采用优先级调度算法 */
00431       if(SEMPRIO == (SEMSCHEDOPTMASK & pstrSem->uiSemOpt))
00432       {
00433           /* 获取信号量表中优先级最高的任务 TCB */
00434           ucTaskPrio = MDS_TaskHighestPrioGet(&pstrSem->strSemTab.strFlag);
00435       }
00436       else /* 信号量采用先进先出调度算法 */
00437       {
00438           /* 采用 0 优先级的链表 */
00439           ucTaskPrio = LOWESTPRIO;
00440       }
00441
00442       pstrNode = MDS_ChainEmpInq(&pstrSem->strSemTab.astrChain[ucTaskPrio]);
00443
00444       /* 信号量中没有被阻塞的任务 */
00445       if(NULL == pstrNode)
00446       {
00447           return (M_TCB*)NULL;
00448       }
00449       else
00450       {
00451           pstrTaskQue = (M_TCBQUE*)pstrNode;
00452
00453           return pstrTaskQue->pstrTcb;
00454       }
00455   }
```

MDS_TaskDelFromSemTab 函数的功能是将任务从信号量调度表中删除，如果信号量采用的是优先级调度方式，则从任务 TCB 中找出任务的相关属性，使用 MDS_TaskDelFromSchedTab 函数将任务从信号量调度表中删除，这个过程与从 ready 表中删除任务的过程是一样的。如果信号量采用的是先进先出的调度方式，则从信号量调度表中删除最先入链表的头节点。函数比较简单，不再详细介绍，代码如下：

```
00388   M_CHAIN* MDS_TaskDelFromSemTab(M_TCB* pstrTcb)
00389   {
00390       M_SEM* pstrSem;
00391       M_CHAIN* pstrChain;
00392       M_PRIOFLAG* pstrPrioFlag;
00393       U8 ucTaskPrio;
00394
00395       /* 获取阻塞任务的信号量 */
00396       pstrSem = pstrTcb->pstrSem;
00397
00398       /* 如果信号量是采用优先级调度算法 */
00399       if(SEMPRIO == (SEMSCHEDOPTMASK & pstrSem->uiSemOpt))
00400       {
00401           /* 获取任务的相关参数 */
00402           ucTaskPrio = pstrTcb->ucTaskPrio;
00403           pstrChain = &pstrSem->strSemTab.astrChain[ucTaskPrio];
00404           pstrPrioFlag = &pstrSem->strSemTab.strFlag;
00405
00406           /* 从 sem 调度表删除当前任务 */
00407           return MDS_TaskDelFromSchedTab(pstrChain, pstrPrioFlag, ucTaskPrio);
00408       }
00409       else /* 如果信号量采用先进先出调度算法 */
00410       {
```

```
00411           /* 获取任务的相关参数，使用 0 优先级链表作为先进现出链表 */
00412           pstrChain = &pstrSem->strSemTab.astrChain[LOWESTPRIO];
00413
00414           /* 从 sem 调度表删除当前任务 */
00415           return MDS_ChainNodeDelete(pstrChain);
00416       }
00417   }
```

MDS_SemGive 函数一次只能释放一个被阻塞的任务，MDS_SemFlush 可以一次性释放被信号量阻塞的所有任务。MDS_SemFlush 函数的原理与 MDS_SemGive 函数是一样的，MDS_SemFlush 函数会循环查找信号量调度表中的所有任务，将他们全部释放掉，不再做详细介绍，代码如下，请读者自行分析：

```
00301   U32 MDS_SemFlush(M_SEM* pstrSem)
00302   {
00303       /* 释放信号量所阻塞的所有任务，被 MDS_SemFlush 释放的阻塞任务返回成功 */
00304       return MDS_SemFlushValue(pstrSem, RTN_SUCD);
00305   }

00224   U32 MDS_SemFlushValue(M_SEM* pstrSem, U32 uiRtnValue)
00225   {
00226       M_TCB* pstrTcb;
00227       M_CHAIN* pstrChain;
00228       M_CHAIN* pstrNode;
00229       M_PRIOFLAG* pstrPrioFlag;
00230       U8 ucTaskPrio;
00231
00232       /* 入口参数检查 */
00233       if(NULL == pstrSem)
00234       {
00235           return RTN_FAIL;
00236       }
00237
00238       (void)MDS_IntLock();
00239
00240       /* 在被该信号量阻塞的任务中获取需要释放的任务 */
00241       while(1)
00242       {
00243           pstrTcb = MDS_SemGetActiveTask(pstrSem);
00244
00245           /* 有阻塞的任务，释放任务 */
00246           if(NULL != pstrTcb)
00247           {
00248               /* 从信号量调度表拆除该任务 */
00249               (void)MDS_TaskDelFromSemTab(pstrTcb);
00250
00251               /* 任务在 delay 表则从 delay 表拆除 */
00252               if(DELAYQUEFLAG == (pstrTcb->uiTaskFlag & DELAYQUEFLAG))
00253               {
00254                   pstrNode = &pstrTcb->strDelayQue.strQueHead;
00255                   (void)MDS_ChainCurNodeDelete(&gstrDelayTab, pstrNode);
00256
00257                   /* 置任务不在 delay 表标志 */
00258                   pstrTcb->uiTaskFlag &= (~((U32)DELAYQUEFLAG));
00259               }
00260
00261               /* 清除任务的 pend 状态 */
```

```
00262               pstrTcb->strTaskOpt.ucTaskSta &= ~((U8)TASKPEND);
00263
00264               /* 借用 uiDelayTick 变量保存 pend 任务的返回值 */
00265               pstrTcb->strTaskOpt.uiDelayTick = uiRtnValue;
00266
00267               /* 将该任务添加到 ready 表中 */
00268               pstrNode = &pstrTcb->strTcbQue.strQueHead;
00269               ucTaskPrio = pstrTcb->ucTaskPrio;
00270               pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00271               pstrPrioFlag = &gstrReadyTab.strFlag;
00272
00273               MDS_TaskAddToSchedTab(pstrChain, pstrNode, pstrPrioFlag, ucTaskPrio);
00274
00275               /* 增加任务的 ready 状态 */
00276               pstrTcb->strTaskOpt.ucTaskSta |= TASKREADY;
00277           }
00278           else /* 没有阻塞的任务，跳出循环操作 */
00279           {
00280               break;
00281           }
00282       }
00283
00284       /* 将信号量置为空 */
00285       pstrSem->uiCounter = SEMEMPTY;
00286
00287       (void)MDS_IntUnlock();
00288
00289       /* 使用软中断调度任务 */
00290       MDS_TaskSwiSched();
00291
00292       return RTN_SUCD;
00293   }
```

当信号量不再需要使用时，可以使用 MDS_SemDelete 函数删除信号量，释放信号量所占用的资源。删除信号量时，被阻塞在该信号量上的所有任务全部会被激活，重新挂入 ready 表中参与任务调度，这个过程使用 MDS_SemFlushValue 函数就可以实现。MDS_SemDelete 函数比较简单，代码如下，不再详细介绍。

```
00313   U32 MDS_SemDelete(M_SEM* pstrSem)
00314   {
00315       /* 入口参数检查 */
00316       if(NULL == pstrSem)
00317       {
00318           return RTN_FAIL;
00319       }
00320
00321       /* 释放信号量所阻塞的所有任务，被 MDS_SemDelete 释放的阻塞任务返回信号量被删除 */
00322       if(RTN_SUCD != MDS_SemFlushValue(pstrSem, RTN_SMTKDL))
00323       {
00324           return RTN_FAIL;
00325       }
00326
00327       return RTN_SUCD;
00328   }
```

本节新增加了一个 pend 任务状态，那么在任务调度的时候也需要对这个状态进行处理，

需要在 MDS_TaskDelayTabSched 函数里增加对 pend 状态的任务的处理，该函数改动较小，只是增加了一个对 pend 状态处理的分支。

```
00544  void MDS_TaskDelayTabSched(void)
00545  {
……                  ……
00588
00589                   /* 如果任务拥有 pend 状态则从 pend 状态恢复 */
00590                   else if(TASKPEND == (TASKPEND& pstrTcb->strTaskOpt.ucTaskSta))
00591                   {
00592                       /* 从信号量调度表拆除任务 */
00593                       (void)MDS_TaskDelFromSemTab(gpstrCurTcb);
00594
00595                       /* 清除任务的 pend 状态 */
00596                       pstrTcb->strTaskOpt.ucTaskSta &= ~((U8)TASKPEND);
00597
00598                       /* 借用 uiDelayTick 变量保存 pend 任务的返回值 */
00599                       pstrTcb->strTaskOpt.uiDelayTick = RTN_SMTKTO;
00600                   }
00601
……                  ……
00631  }
```

00590 行，走到此处说明 delay 表中有时间耗尽的任务，如果这个任务是 pend 状态，那么走此分支。

00593 行，走到此处说明该任务是 pend 状态，并且时间耗尽，需要从信号量调度表删除。

00596 行，清除任务的 pend 状态。

00599 行，将返回值 RTN_SMTKTO 存入超时任务的 TCB 中的 strTaskOpt.uiDelayTick 变量中，当这个超时任务返回时，返回值 RTN_SMTKTO 可以说明这个任务是由于时间耗尽超时返回的。

任务增加了 pend 状态，还影响到了 MDS_TaskDelete 函数，在删除处于 pend 状态的任务时需要将这个任务从相关的信号量调度表中拆除，MDS_TaskDelete 函数新增的代码如下：

```
00121  U32 MDS_TaskDelete(M_TCB* pstrTcb)
00122  {
……      ……
00167
00168       /* 任务拥有 pend 状态 */
00169       if(TASKPEND == (TASKPEND & ucTaskSta))
00170       {
00171           /* 从信号量调度表拆除任务 */
00172           (void)MDS_TaskDelFromSemTab(pstrTcb);
00173       }
00174
……      ……
00191  }
```

当我们需要串行访问一个需要多步骤操作的资源时，就可以使用二进制信号量对其进行保护，例如对串口、FLASH 等外设操作工程的保护。在使用信号量保护资源时要避免出现信号量死锁的情况，所谓信号量死锁就是指每个任务占有一个信号量，它们都因无法获取到对方已占有的信号量而无法运行，因此本身占有的信号量也无法被释放，这样，多个任务之

间形成互相等待信号量的情况，所有任务都处于一种互相等待的永久等待状态，形成死锁。例如任务 A 已经获取到了信号量 a，任务 B 已经获取到了信号量 b，但任务 A 需要获取到信号量 b 才能继续运行，而任务 B 则需要获取到信号量 a 才能继续运行，这样 2 个任务就会形成死锁，这两个任务也就永远无法运行了。

```
void TaskA(void)
{
    已获取信号量 a;

    (void)MDS_SemTake(&b, SEMWAITFEV);

    ……
}
```

```
void TaskB(void)
{
    已获取信号量 b;

    (void)MDS_SemTake(&a, SEMWAITFEV);

    ……
}
```

前面我们主要介绍了二进制信号量互斥的作用，二进制信号量互斥的这个特点也可以用来触发任务，当同步事件使用。例如，当我们需要由任务 B 来触发任务 A 的运行时，先将信号量 a 初始化为空状态，由任务 A 获取信号量 a，由任务 B 释放 a。当任务 B 没有释放信号量 a 时，任务 A 就会因为获取不到信号量 a 而被阻塞，一旦任务 B 释放了信号量 a，任务 B 就会被激活，开始运行。

```
void TaskA(void)
{
    ……

(void)MDS_SemTake(&a, SEMWAITFEV);

    ……
}
```

```
void TaskB(void)
{
    ……

    (void)MDS_SemGive(&a);

    ……
}
```

到目前为止操作系统已经有了 ready 表、delay 表和多个信号量表，系统运行时需要不断的更新任务 TCB 中的结构与各种调度表之间的关系。strTcbQue 结构中的 strQueHead 节点可以挂接到 ready 表或者信号量调度表，strDelayQue 结构中的 strQueHead 节点可以挂接到 delay 表，pstrSem 指针需要指向阻塞任务的信号量。

图 53 列出了多个不同状态的任务与这些调度表之间的关系，任务 1 被信号量 sem1 阻塞，处于非永久 pend 态，strDelayQue 结构中的 strQueHead 节点被挂接到 delay 表，strTcbQue 结构中的 strQueHead 节点被挂接到 sem1 表，pstrSem 指针指向 sem1。任务 2 处于 ready 态，strTcbQue 结构中的 strQueHead 节点被挂接到 ready 表。任务 3 处于非永久 delay 态，strDelayQue 结构中的 strQueHead 节点被挂接到 delay 表。任务 4 被信号量 sem1 阻塞，处于永久 pend 态，strTcbQue 结构中的 strQueHead 节点被挂接到 sem1 表，pstrSem 指针指向 sem1。任务 5 被信号量 sem2 阻塞，处于非永久 pend 态，strDelayQue 结构中的 strQueHead 节点被挂接到 delay 表，strTcbQue 结构中的 strQueHead 节点被挂接到 sem2 表，pstrSem 指针指向 sem2。任务 6 处于永久 delay 态，不与任何调度表有关系。

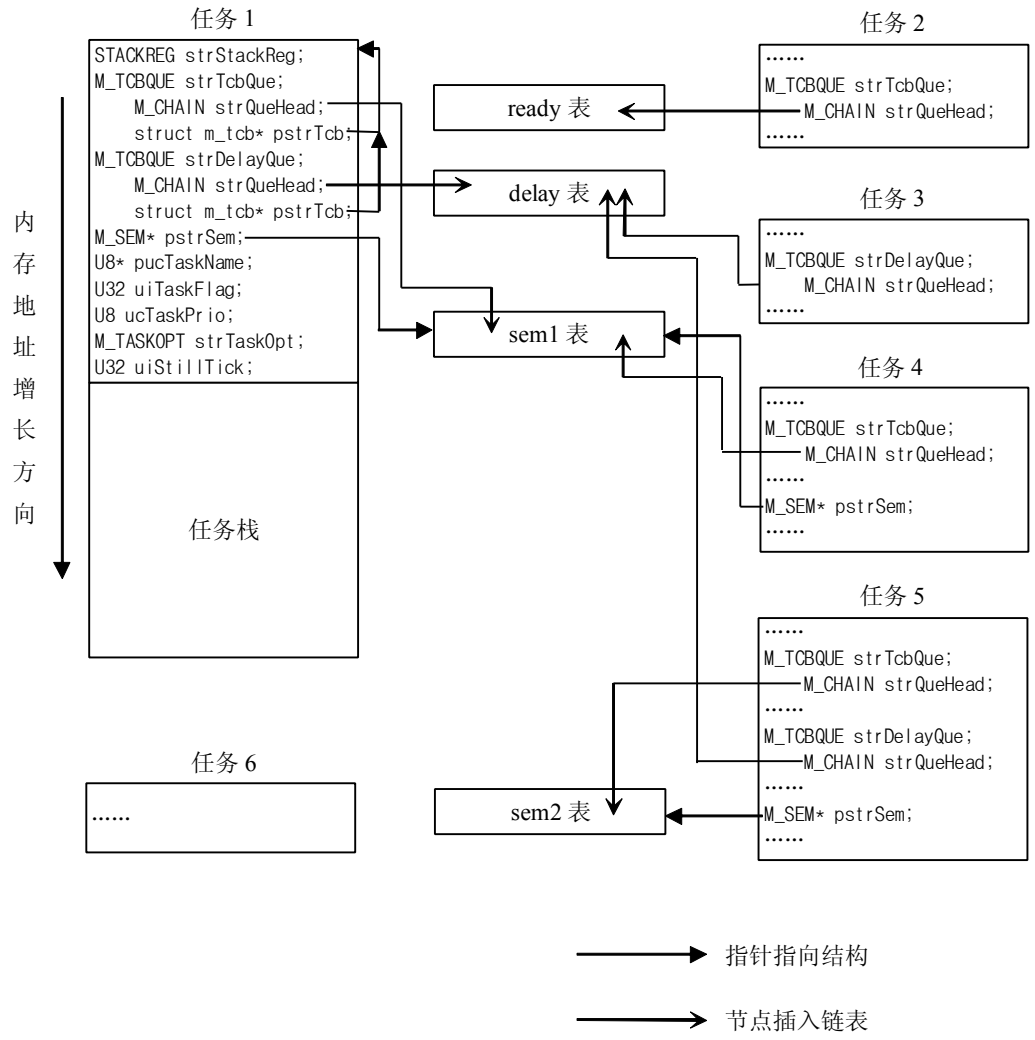


图 53　TCB 与各种调度表的关系

以上是本节新增加的内容，本节使用了 4 个验证任务 TEST_TestTask1~TEST_TestTask4，它们的优先级分别为 6，2，4，5，还有 2 个信号量 gstrSemSync 和 gstrSemMute，它们都被初始化为 FIFO 类型。

TEST_TestTask3 和 TEST_TestTask4 任务在开始的前 3 次循环中为互斥任务，在 TEST_TestTask3 任务获取到 gstrSemMute 信号量的期间内，TEST_TestTask4 任务被 gstrSemMute 信号量阻塞，当 TEST_TestTask3 任务释放 gstrSemMute 信号量后 TEST_TestTask4 任务获取到 gstrSemMute 信号量并开始运行，此时 TEST_TestTask3 任务会被 gstrSemMute 信号量阻塞，直到 TEST_TestTask4 任务再次释放 gstrSemMute 信号量，这两个任务如此循环 3 次。3 次循环之后，这两个任务都被阻塞到 gstrSemSync 信号量，需要由 TEST_TestTask1 任务释放 gstrSemSync 信号量来触发这 2 个任务的运行。

TEST_TestTask1 任务的前 10 次循环用来释放 gstrSemSync 信号量，触发被阻塞到这个信号量上的任务。TEST_TestTask2 任务一直在获取 gstrSemSync 信号量，需要由 TEST_TestTask1 任务激活。TEST_TestTask1 任务运行 10 次循环之后，接下来的 5 次循环使用 MDS_SemFlush 函数激活所有被阻塞到 gstrSemSync 信号量上的任务，5 次循环之后，TEST_TestTask1 任务会删除 gstrSemSync 信号量，另外三个任务检测到 gstrSemSync 信号量被删除，会退出主函数，结束任务的运行。

```
00021  void TEST_TestTask1(void)
00022  {
00023      U8 i;
00024
00025      i = 0;
00026
00027      while(1)
00028      {
00029          DEV_PutStrToMem((U8*)"\r\nTask1 is running ! Tick is: %d",
00030                          MDS_SystemTickGet());
00031
00032          DEV_DelayMs(1500);
00033
00034          (void)MDS_TaskDelay(200);
00035
00036          /* 前10次, 每次运行释放一次gstrSemSync信号量 */
00037          if(i < 10)
00038          {
00039              i++;
00040
00041              DEV_PutStrToMem((U8*)"\r\nTask1 give gstrSemSync %d! Tick is: %d",
00042                              i, MDS_SystemTickGet());
00043
00044              /* 同步其它任务 */
00045              (void)MDS_SemGive(&gstrSemSync);
00046          }
00047          /* 接下来5次, 每次运行flush一次gstrSemSync信号量 */
00048          else if(i < 15)
00049          {
00050              i++;
00051
00052              DEV_PutStrToMem((U8*)"\r\nTask1 flush gstrSemSync %d! Tick is: %d", i,
00053                              MDS_SystemTickGet());
00054
00055              /* 释放所有被gstrSemSync阻塞的任务 */
00056              (void)MDS_SemFlush(&gstrSemSync);
00057          }
00058          /* 删除gstrSemSync信号量 */
00059          else if(15 == i)
00060          {
00061              i++;
00062
00063              DEV_PutStrToMem((U8*)"\r\nTask1 delete gstrSemSync %d! Tick is: %d", i,
00064                              MDS_SystemTickGet());
00065
00066              /* 删除gstrSemSync信号量 */
00067              (void)MDS_SemDelete(&gstrSemSync);
00068          }
00069      }
00070  }

00077  void TEST_TestTask2(void)
00078  {
00079      U8 i;
00080
00081      i = 0;
00082
00083      while(1)
00084      {
```

```
00085         /* 信号量被删除，任务返回 */
00086         if(RTN_SMTKDL == MDS_SemTake(&gstrSemSync, SEMWAITFEV))
00087         {
00088             DEV_PutStrToMem((U8*)"\r\nTask2 gstrSemSync deleted! Tick is: %d",
00089                             MDS_SystemTickGet());
00090 
00091             return;
00092         }
00093         else /* 获取到gstrSemSync信号量才运行 */
00094         {
00095             i++;
00096 
00097             DEV_PutStrToMem((U8*)"\r\nTask2 take gstrSemSync %d! Tick is: %d", i,
00098                             MDS_SystemTickGet());
00099 
00100             DEV_DelayMs(100);
00101         }
00102     }
00103 }

00110 void TEST_TestTask3(void)
00111 {
00112     U8 i;
00113 
00114     i = 0;
00115 
00116     while(1)
00117     {
00118         /* 前3次获取gstrSemMute信号量，与TEST_TestTask4任务互锁 */
00119         if(i < 3)
00120         {
00121             i++;
00122 
00123             /* 获取到信号量才运行 */
00124             (void)MDS_SemTake(&gstrSemMute, SEMWAITFEV);
00125 
00126             DEV_PutStrToMem((U8*)"\r\nTask3 take gstrSemMute %d! Tick is: %d", i,
00127                             MDS_SystemTickGet());
00128 
00129             DEV_DelayMs(500);
00130 
00131             (void)MDS_TaskDelay(150);
00132 
00133             DEV_PutStrToMem((U8*)"\r\nTask3 give gstrSemMute %d! Tick is: %d", i,
00134                             MDS_SystemTickGet());
00135 
00136             /* 释放信号量，以便其它任务可以获得该信号量 */
00137             (void)MDS_SemGive(&gstrSemMute);
00138         }
00139         else /* 接下来获取gstrSemSync信号量，由TEST_TestTask1任务激活 */
00140         {
00141             i++;
00142 
00143             /* 信号量被删除，任务返回 */
00144             if(RTN_SMTKDL == MDS_SemTake(&gstrSemSync, SEMWAITFEV))
00145             {
00146                 DEV_PutStrToMem((U8*)"\r\nTask3 gstrSemSync deleted! Tick is: %d",
00147                                 MDS_SystemTickGet());
00148 
```

```
00149                 return;
00150             }
00151             else /* 获取到 gstrSemSync 信号量才运行 */
00152             {
00153                 DEV_PutStrToMem((U8*)"\r\nTask3 take gstrSemSync %d! Tick is: %d",
00154                                 i, MDS_SystemTickGet());
00155 
00156                 DEV_DelayMs(500);
00157 
00158                 (void)MDS_TaskDelay(150);
00159             }
00160         }
00161     }
00162 }

00169 void TEST_TestTask4(void)
00170 {
00171     U8 i;
00172 
00173     i = 0;
00174 
00175     while(1)
00176     {
00177         /* 前 3 次获取 gstrSemMute 信号量, 与 TEST_TestTask4 任务互锁 */
00178         if(i < 3)
00179         {
00180             i++;
00181 
00182             /* 获取到信号量才运行 */
00183             (void)MDS_SemTake(&gstrSemMute, SEMWAITFEV);
00184 
00185             DEV_PutStrToMem((U8*)"\r\nTask4 take gstrSemMute %d! Tick is: %d", i,
00186                             MDS_SystemTickGet());
00187 
00188             DEV_DelayMs(500);
00189 
00190             (void)MDS_TaskDelay(200);
00191 
00192             DEV_PutStrToMem((U8*)"\r\nTask4 give gstrSemMute %d! Tick is: %d", i,
00193                             MDS_SystemTickGet());
00194 
00195             /* 释放信号量, 以便其它任务可以获得该信号量 */
00196             (void)MDS_SemGive(&gstrSemMute);
00197         }
00198         else /* 接下来获取 gstrSemSync 信号量, 由 TEST_TestTask1 任务激活 */
00199         {
00200             i++;
00201 
00202             /* 信号量被删除, 任务返回 */
00203             if(RTN_SMTKDL == MDS_SemTake(&gstrSemSync, SEMWAITFEV))
00204             {
00205                 DEV_PutStrToMem((U8*)"\r\nTask4 gstrSemSync deleted! Tick is: %d",
00206                                 MDS_SystemTickGet());
00207 
00208                 return;
00209             }
00210             else /* 获取到 gstrSemSync 信号量才运行 */
00211             {
00212                 DEV_PutStrToMem((U8*)"\r\nTask4 take gstrSemSync %d! Tick is: %d",
```

```
00213                                   i, MDS_SystemTickGet());
00214
00215                   DEV_DelayMs(500);
00216
00217                   (void)MDS_TaskDelay(200);
00218               }
00219           }
00220       }
00221   }
```

按照这 4 个任务设定的循环时间和任务优先级，开始时应该会看到 TEST_TestTask1 任务触发 TEST_TestTask2 任务，同时 TEST_TestTask3 任务和 TEST_TestTask4 任务互斥运行。当 TEST_TestTask3 任务和 TEST_TestTask4 任务互斥运行 3 次之后，它们不再互斥运行，而是变为同 TEST_TestTask2 任务一样，需要由 TEST_TestTask1 任务激活，此时应该看到 TEST_TestTask1 任务会顺序激活其它 3 个任务。当 TEST_TestTask1 任务运行 10 个循环之后，会使用 MDS_SemFlush 函数同时激活所有其它 3 个任务，应该会看到这 3 个任务全部被激活运行。TEST_TestTask1 任务执行 MDS_SemFlush 函数 5 次之后会删除同步信号量，其它 3 个任务会发现信号量被删除而退出任务。

本节运行结果截图如下：

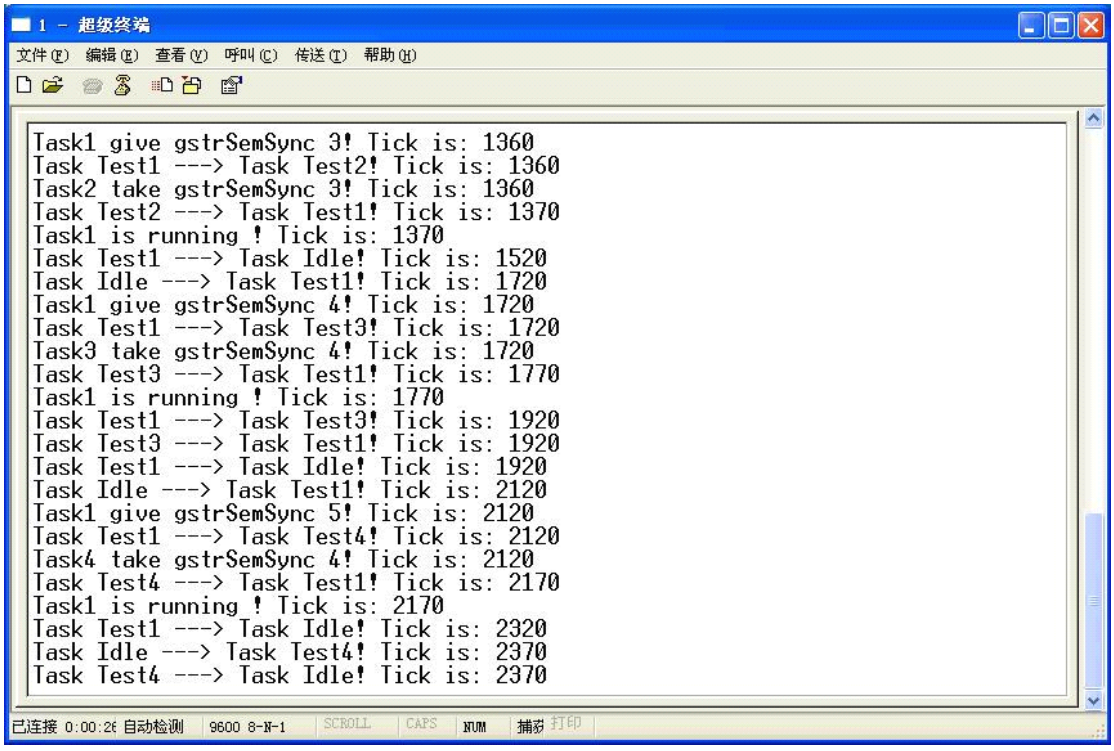


图 54　任务获取信号量的打印信息

读者可以访问 http://blog.sina.com.cn/ifreecoding 网站下载视频，观看全部数据的打印过程，从打印数据可以看到实际输出结果与我们的设计是一致的。

使用工具软件解析本节任务切换过程的数据，结果如下图所示：

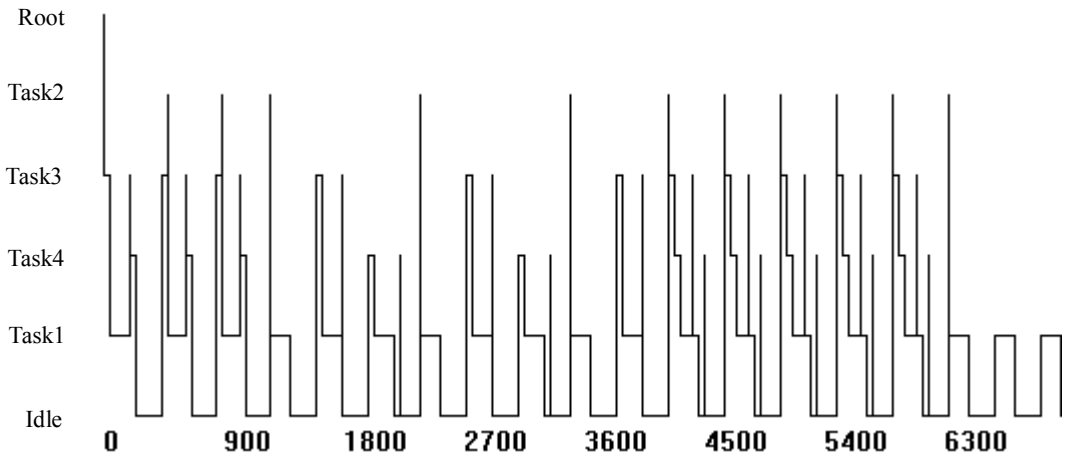


图 55　4.7 节任务切换过程图

为了对比二进制信号量 FIFO 和 PRIO 属性的区别，我们将 gstrSemSync 信号量由 FIFO 属性改为 PRIO 属性，重新编译运行输出结果，解析 PRIO 属性的输出数据，结果如图 56 所示：

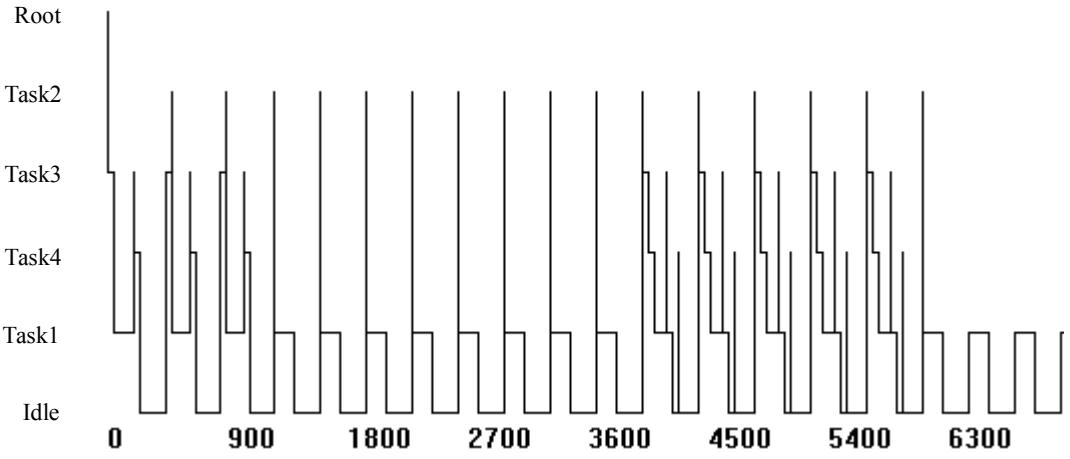


图 56　信号量改为 PRIO 属性后的任务切换过程图

我们对比图 55 和图 56 可以发现中间部分有明显的区别，中间部分是由 TEST_TestTask1 任务释放信号量激活其它 3 个任务的过程。在图 55 中，TEST_TestTask1 任务释放信号量之后，TEST_TestTask2、TEST_TestTask3 和 TEST_TestTask4 任务轮流被激活，而在图 56 中，TEST_TestTask1 任务释放信号量之后却只有 TEST_TestTask2 任务被激活，这是因为图 55 的信号量采用的是 FIFO 属性的，会按照信号量阻塞任务的顺序轮流激活任务，而图 56 的信号量采用的是 PRIO 属性的，会激活被信号量所阻塞任务中的优先级最高的任务，因此只有优先级最高的 TEST_TestTask2 任务会被激活。

第 8 节 计数信号量

在上一节我们了解了信号量的原理，也使用该原理编写了代码，实现了二进制信号量的功能，本节我们将实现计数信号量的功能。对比二进制信号量，计数信号量可以实现对信号量的累计计数，记录释放放信号量的所有次数，因此也可以使任务获取到等量的信号量次数。

如果当二进制信号量处于满状态时再去释放信号量，那么二进制信号量仍会处于满状态，这次释放信号量的动作不会产生任何影响。计数信号量则可以记录信号量释放、获取的次数，每释放一次信号量，计数信号量里相应的统计变量就会加 1，相反，任务每获取一次信号量，计数信号量里相应的统计变量就会减 1。

看下表，二进制信号量和计数信号量的对比：

操作方式	二进制信号量	计数信号量
MDS_SemCreate	满，信号量被初始化为满状态。	1，信号量计数值被初始化为 1。
MDS_SemTake	空，任务获取到信号量。	空，任务获取到信号量，信号量计数值被置为 0。
MDS_SemTake	空，任务没有获取到信号量，被阻塞。	空，任务没有获取到信号量，被阻塞，信号量计数值仍为 0。
MDS_SemTake	空，任务没有获取到信号量，共有 2 个任务被阻塞。	空，任务没有获取到信号量，共有 2 个任务被阻塞，信号量计数值仍为 0。
MDS_SemGive	空，一个任务获取到信号量，重新恢复到 ready 态，还有一个任务被阻塞。	空，一个任务获取到信号量，重新恢复到 ready 态，还有一个任务被阻塞，信号量计数值仍为 0。
MDS_SemGive	空，一个任务获取到信号量，重新恢复到 ready 态，没有任务被阻塞。	空，一个任务获取到信号量，重新恢复到 ready 态，没有任务被阻塞，信号量计数值仍为 0。
MDS_SemGive	满，没有任务获取信号量，信号量被置为满状态。	1，没有任务获取信号量，信号量计数值被置为 1。
MDS_SemGive	满，没有任务获取信号量，信号量仍为满状态。	2，没有任务获取信号量，信号量计数值被置为 2。
MDS_SemGive	满，没有任务获取信号量，信号量仍为满状态。	3，没有任务获取信号量，信号量计数值被置为 3。
MDS_SemTake	空，任务获取到信号量。	2，一个任务获取到信号量，信号量计数值被置为 2。
MDS_SemTake	空，任务没有获取到信号量，被阻塞。	1，一个任务获取到信号量，信号量计数值被置为 1。
MDS_SemTake	空，任务没有获取到信号量，共有 2 个任务被阻塞。	空，一个任务获取到信号量，信号量计数值被置为 0。
MDS_SemTake	空，任务没有获取到信号量，共有 3 个任务被阻塞。	空，任务没有获取到信号量，被阻塞，信号量计数值仍为 0。

表 10　二进制信号量与计数信号量的对比

从上面的介绍可以看出，只需要在二进制信号量的基础上增加一个计数值就可以实现计数信号量的功能，我们在 4.7 节中已经定义了信号量的结构体，如下：

```
typedef struct m_sem
{
    M_TASKSCHEDTAB strSemTab;       /* 信号量调度表 */
    U32 uiCounter;                  /* 信号量计数值 */
    U32 uiSemOpt;                   /* 信号量参数 */
}M_SEM;
```

在二进制信号量中，uiCounter 变量为 0 时表示信号量处于空状态，为 0xFFFFFFFF 时表示信号量为满状态，在计数信号量里正好也可以使用该变量表示计数信号量的值，非空非满的数值就是计数信号量的计数数值。为此，我们需要对 MDS_SemCreate、MDS_SemTake 和 MDS_SemGive 这 3 个与信号量操作有关的函数做一些简单的修改，修改的原则就是依据表 10 来实现的，将原有的二进制信号量的空满状态扩展一下。

新增的代码没有本质的变化，比较简单，仅列出，不再做详细解释，读者可以使用 Beyond Compare 等代码对比工具对比查看本节代码与 4.7 节代码的异同。

```
00023  U32 MDS_SemCreate(M_SEM* pstrSem, U32 uiSemOpt, U32 uiInitVal)
00024  {
00025      /* 入口参数检查 */
00026      if(NULL == pstrSem)
00027      {
00028          return RTN_FAIL;
00029      }
00030
00031      /* 信号量选项检查 */
00032      if(((SEMBIN != (SEMTYPEMASK & uiSemOpt))
00033          && (SEMCNT != (SEMTYPEMASK & uiSemOpt)))
00034         || ((SEMFIFO != (SEMSCHEDOPTMASK & uiSemOpt))
00035             && (SEMPRIO != (SEMSCHEDOPTMASK & uiSemOpt))))
00036      {
00037          return RTN_FAIL;
00038      }
00039
00040      /* 二进制信号量初始值只能是空或者满 */
00041      if(SEMBIN == (SEMTYPEMASK & uiSemOpt))
00042      {
00043          if((SEMEMPTY != uiInitVal) && (SEMFULL != uiInitVal))
00044          {
00045              return RTN_FAIL;
00046          }
00047      }
00048
00049      /* 初始化信号量调度表 */
00050      MDS_TaskSchedTabInit(&pstrSem->strSemTab);
00051
00052      /* 初始化信号量初始值 */
00053      pstrSem->uiCounter = uiInitVal;
00054
00055      /* 初始化信号量参数 */
00056      pstrSem->uiSemOpt = uiSemOpt;
00057
00058      return RTN_SUCD;
00059  }
```

```
00078  U32 MDS_SemTake(M_SEM* pstrSem, U32 uiDelayTick)
00079  {
00080      /* 入口参数检查 */
00081      if(NULL == pstrSem)
00082      {
00083          return RTN_FAIL;
00084      }
00085
00086      (void)MDS_IntLock();
00087
```

```
00088       /* 更新与当前任务相关的信号量 */
00089       gpstrCurTcb->pstrSem = pstrSem;
00090
00091       /* 获取信号量时不等待时间 */
00092       if(SEMNOWAIT == uiDelayTick)
00093       {
00094           /* 二进制信号量 */
00095           if(SEMBIN == (SEMTYPEMASK & pstrSem->uiSemOpt))
00096           {
00097               /* 信号量为满, 可获取到信号量 */
00098               if(SEMFULL == pstrSem->uiCounter)
00099               {
00100                   /* 获取到信号量后将信号量置为空 */
00101                   pstrSem->uiCounter = SEMEMPTY;
00102
00103                   (void)MDS_IntUnlock();
00104
00105                   return RTN_SUCD;
00106               }
00107               else /* 信号量为空, 无法获取到信号量 */
00108               {
00109                   (void)MDS_IntUnlock();
00110
00111                   return RTN_SMTKRT;
00112               }
00113           }
00114           else /* 计数信号量 */
00115           {
00116               /* 信号量不为空, 可获取到信号量 */
00117               if(SEMEMPTY != pstrSem->uiCounter)
00118               {
00119                   /* 获取到信号量后将信号量计数值-1 */
00120                   pstrSem->uiCounter--;
00121
00122                   (void)MDS_IntUnlock();
00123
00124                   return RTN_SUCD;
00125               }
00126               else /* 信号量为空, 无法获取到信号量 */
00127               {
00128                   (void)MDS_IntUnlock();
00129
00130                   return RTN_SMTKRT;
00131               }
00132           }
00133       }
00134       else /* 获取信号量时需要等待时间 */
00135       {
00136           /* 二进制信号量 */
00137           if(SEMBIN == (SEMTYPEMASK & pstrSem->uiSemOpt))
00138           {
00139               /* 信号量为满, 可获取到信号量 */
00140               if(SEMFULL == pstrSem->uiCounter)
00141               {
00142                   /* 获取到信号量后将信号量置为空 */
00143                   pstrSem->uiCounter = SEMEMPTY;
00144
00145                   (void)MDS_IntUnlock();
00146
```

```
00147                 return RTN_SUCD;
00148             }
00149             else /* 信号量为空，无法获取到信号量，需要切换任务 */
00150             {
00151                 /* 将任务置为 pend 状态 */
00152                 if(RTN_FAIL == MDS_TaskPend(pstrSem, uiDelayTick))
00153                 {
00154                     (void)MDS_IntUnlock();
00155 
00156                     /* 任务 pend 失败 */
00157                     return RTN_FAIL;
00158                 }
00159 
00160                 (void)MDS_IntUnlock();
00161 
00162                 /* 使用软中断调度任务 */
00163                 MDS_TaskSwiSched();
00164 
00165                 /* 任务 pend 返回值，该值在任务 pend 状态结束时被保存在 uiDelayTick 中 */
00166                 return gpstrCurTcb->strTaskOpt.uiDelayTick;
00167             }
00168         }
00169         else /* 计数信号量 */
00170         {
00171             /* 信号量不为空，可获取到信号量 */
00172             if(SEMEMPTY != pstrSem->uiCounter)
00173             {
00174                 /* 获取到信号量后将信号量计数值-1 */
00175                 pstrSem->uiCounter--;
00176 
00177                 (void)MDS_IntUnlock();
00178 
00179                 return RTN_SUCD;
00180             }
00181             else /* 信号量为空，无法获取到信号量，需要切换任务 */
00182             {
00183                 /* 将任务置为 pend 状态 */
00184                 if(RTN_FAIL == MDS_TaskPend(pstrSem, uiDelayTick))
00185                 {
00186                     (void)MDS_IntUnlock();
00187 
00188                     /* 任务 pend 失败 */
00189                     return RTN_FAIL;
00190                 }
00191 
00192                 (void)MDS_IntUnlock();
00193 
00194                 /* 使用软中断调度任务 */
00195                 MDS_TaskSwiSched();
00196 
00197                 /* 任务 pend 返回值，该值在任务 pend 状态结束时被保存在 uiDelayTick 中 */
00198                 return gpstrCurTcb->strTaskOpt.uiDelayTick;
00199             }
00200         }
00201     }
00202 }

00211 U32 MDS_SemGive(M_SEM* pstrSem)
00212 {
```

```
00213       M_TCB* pstrTcb;
00214       M_CHAIN* pstrChain;
00215       M_CHAIN* pstrNode;
00216       M_PRIOFLAG* pstrPrioFlag;
00217       U32 uiRtn;
00218       U8 ucTaskPrio;
00219
00220       /* 入口参数检查 */
00221       if(NULL == pstrSem)
00222       {
00223           return RTN_FAIL;
00224       }
00225
00226       uiRtn = RTN_SUCD;
00227
00228       (void)MDS_IntLock();
00229
00230       /* 信号量为空 */
00231       if(SEMEMPTY == pstrSem->uiCounter)
00232       {
00233           /* 在被该信号量阻塞的任务中获取需要释放的任务 */
00234           pstrTcb = MDS_SemGetActiveTask(pstrSem);
00235
00236           /* 有阻塞的任务, 释放任务 */
00237           if(NULL != pstrTcb)
00238           {
00239               /* 从信号量调度表拆除该任务 */
00240               (void)MDS_TaskDelFromSemTab(pstrTcb);
00241
00242               /* 任务在 delay 表则从 delay 表拆除 */
00243               if(DELAYQUEFLAG == (pstrTcb->uiTaskFlag & DELAYQUEFLAG))
00244               {
00245                   pstrNode = &pstrTcb->strDelayQue.strQueHead;
00246                   (void)MDS_ChainCurNodeDelete(&gstrDelayTab, pstrNode);
00247
00248                   /* 置任务不在 delay 表标志 */
00249                   pstrTcb->uiTaskFlag &= (~((U32)DELAYQUEFLAG));
00250               }
00251
00252               /* 清除任务的 pend 状态 */
00253               pstrTcb->strTaskOpt.ucTaskSta &= ~((U8)TASKPEND);
00254
00255               /* 借用 uiDelayTick 变量保存 pend 任务的返回值 */
00256               pstrTcb->strTaskOpt.uiDelayTick = RTN_SUCD;
00257
00258               /* 将该任务添加到 ready 表中 */
00259               pstrNode = &pstrTcb->strTcbQue.strQueHead;
00260               ucTaskPrio = pstrTcb->ucTaskPrio;
00261               pstrChain = &gstrReadyTab.astrChain[ucTaskPrio];
00262               pstrPrioFlag = &gstrReadyTab.strFlag;
00263
00264               MDS_TaskAddToSchedTab(pstrChain, pstrNode, pstrPrioFlag, ucTaskPrio);
00265
00266               /* 增加任务的 ready 状态 */
00267               pstrTcb->strTaskOpt.ucTaskSta |= TASKREADY;
00268
00269               (void)MDS_IntUnlock();
00270
00271               /* 使用软中断调度任务 */
```

```
00272             MDS_TaskSwiSched();
00273
00274             return uiRtn;
00275         }
00276         else /* 没有阻塞的任务 */
00277         {
00278             /* 二进制信号量 */
00279             if(SEMBIN == (SEMTYPEMASK & pstrSem->uiSemOpt))
00280             {
00281                 /* 释放信号量后将信号量置为满 */
00282                 pstrSem->uiCounter = SEMFULL;
00283             }
00284             else /* 计数信号量 */
00285             {
00286                 /* 释放信号量后将信号量+1. 走此分支信号量+1 不会溢出 */
00287                 pstrSem->uiCounter++;
00288             }
00289         }
00290     }
00291     else /* 信号量非空 */
00292     {
00293         /* 计数信号量 */
00294         if(SEMCNT == (SEMTYPEMASK & pstrSem->uiSemOpt))
00295         {
00296             /* 信号量未满 */
00297             if(SEMFULL != pstrSem->uiCounter)
00298             {
00299                 /* 释放信号量后将信号量+1 */
00300                 pstrSem->uiCounter++;
00301             }
00302             else /* 信号量已满 */
00303             {
00304                 uiRtn = RTN_SMGVOV;
00305             }
00306         }
00307     }
00308
00309     (void)MDS_IntUnlock();
00310
00311     return uiRtn;
00312 }
```

本节设计了 3 个任务来验证计数信号量的功能，TEST_TestTask1 任务先运行 1000ms，然后连续释放 4 次计数信号量，之后延迟 500 个 ticks，之后不断重复上述操作。TEST_TestTask2 任务获取一次计数信号量之后延迟 200 个 ticks，再重复上述操作。TEST_TestTask3 任务每次获取计数信号量后会有一个短暂的 10 个 ticks 的延迟时间，不断重复这个过程。

```
00019 void TEST_TestTask1(void)
00020 {
00021     while(1)
00022     {
00023         DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00024                         MDS_SystemTickGet());
00025
00026         DEV_DelayMs(1000);
00027
```

```
00028          /* 连续释放计数信号量, 激活其它任务 */
00029          (void)MDS_SemGive(&gstrSemCnt);
00030          (void)MDS_SemGive(&gstrSemCnt);
00031          (void)MDS_SemGive(&gstrSemCnt);
00032          (void)MDS_SemGive(&gstrSemCnt);
00033
00034          (void)MDS_TaskDelay(500);
00035      }
00036  }
```

```
00043  void TEST_TestTask2(void)
00044  {
00045      while(1)
00046      {
00047          /* 获取到信号量才运行 */
00048          (void)MDS_SemTake(&gstrSemCnt, SEMWAITFEV);
00049
00050          DEV_PutStrToMem((U8*)"\r\nTask2 is running! Tick is: %d",
00051                          MDS_SystemTickGet());
00052
00053          (void)MDS_TaskDelay(200);
00054      }
00055  }
```

```
00062  void TEST_TestTask3(void)
00063  {
00064      while(1)
00065      {
00066          /* 获取到信号量才运行 */
00067          (void)MDS_SemTake(&gstrSemCnt, SEMWAITFEV);
00068
00069          DEV_PutStrToMem((U8*)"\r\nTask3 is running! Tick is: %d",
00070                          MDS_SystemTickGet());
00071
00072          (void)MDS_TaskDelay(10);
00073      }
00074  }
```

信号量被初始化为空状态的计数信号量，TEST_TestTask1 任务优先级被初始化为 2，TEST_TestTask2 任务优先级被初始化为 1，TEST_TestTask3 任务优先级被初始化为 3。系统运行时 TEST_TestTask2 任务最先开始运行,会被阻塞在计数信号量上,然后 TEST_TestTask1 任务开始运行，释放计数信号量激活 TEST_TestTask2 任务，TEST_TestTask2 任务获取到信号量之后进入延迟状态，TEST_TestTask1 任务又连续释放了 3 次计数信号量之后进入延迟状态，然后 TEST_TestTask3 任务开始运行，连续获取到 3 次计数信号量之后被阻塞到该信号量上。当 TEST_TestTask2 任务重新运行时也会被阻塞到计数信号量上，然后 TEST_TestTask1 任务又开始运行，不断重复上述过程。

下图是实际运行结果的截图,读者可以到 http://blog.sina.com.cn/ifreecoding 网站下载视频观看动态输出过程。

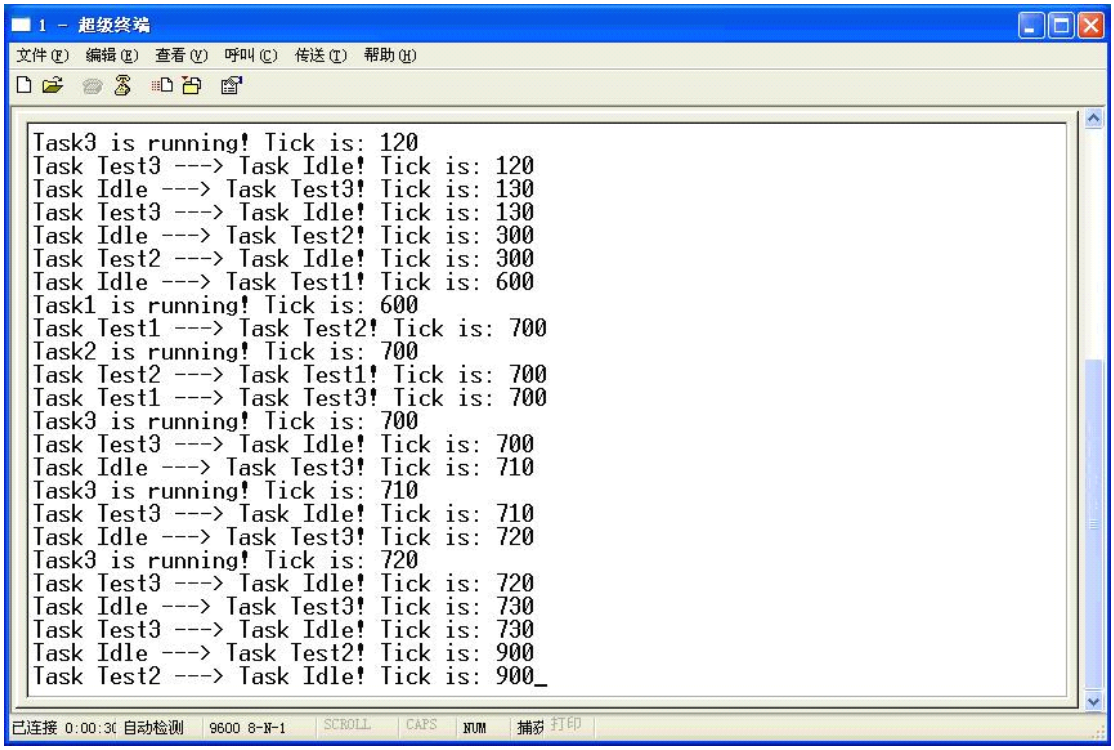


图 57　计数信号量的打印信息

使用工具软件解析本节任务切换过程的数据，结果如下图所示：

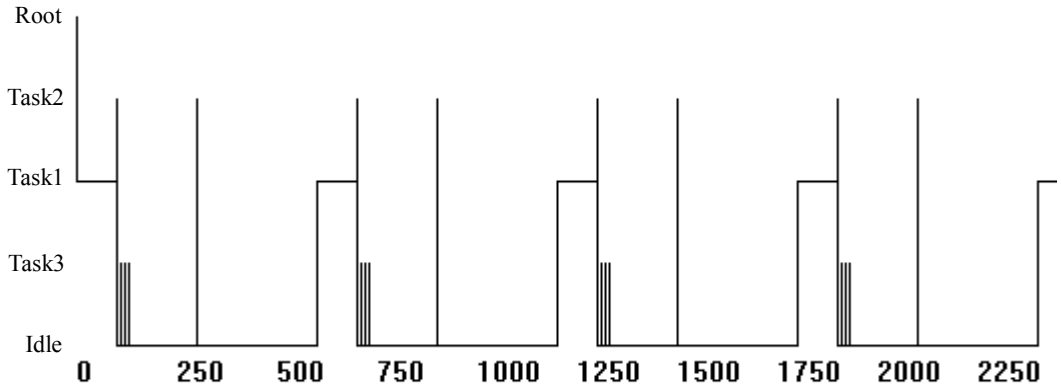


图 58　4.8 节任务切换过程图

可以看出这几个任务切换的过程与我们的设计是一致的。

第 9 节 互斥信号量

前面 2 节我们实现了二进制信号量和计数信号量，本节我们将实现最后一种信号量——互斥信号量。互斥信号量与二进制信号量一样也只有空满两种状态，但互斥信号量要比二进制更复杂，具有更多的功能。

我们使用信号量保护资源串行执行时，在某些函数里可能会发生信号量嵌套应用，这种情况下使用二进制信号量或者计数信号量就比较难处理，下面我们以对 FLASH 操作的例子

来进行说明。

假设我们需要提供 2 种 FLASH 擦除函数，一种是块擦除函数 FlashBlockErase，该函数的入口参数是 FLASH 的块号，每调用一次该函数就擦除 FLASH 中的一个块的数据。另一种是 FLASH 区域擦除函数 FlashErase，该函数的入口参数是需要擦除的 FLASH 起始和结束地址，调用该函数时，根据需要擦除的 FLASH 地址范围计算出 FLASH 的块号，然后在 for 循环中调用 FlashBlockErase 函数依次擦除 FLASH 中各个块的数据即可实现 FlashErase 函数的功能。这两个函数都作为接口函数对外提供，需要防止多任务对 FLASH 操作的重入，因此需要使用信号量对 FLASH 的操作过程进行保护，看下面伪码：

```
FlashBlockErase(BlockNum)
{
    MDS_SemTake(FLASH 信号量);

    操作 FLASH 硬件寄存器擦除 FLASH 块数据;

    MDS_SemGive(FLASH 信号量);
}

FlashErase(StartAddr, EndAddr)
{
    计算 StartAddr 和 EndAddr 之间所占用的 FLASH 块;

    MDS_SemTake(FLASH 信号量);

    /* 调用 FlashBlockErase 函数循环擦除 FLASH 块数据 */
    for()
    {
        FlashBlockErase();
    }

    MDS_SemGive(FLASH 信号量);
}
```

当我们调用 FlashErase 函数时，仅从信号量的角度来看看函数的执行过程：

```
00001  FlashErase(StartAddr, EndAddr)
00002  {
00003      MDS_SemTake(FLASH 信号量);
00004
00005      for()
00006      {
00007          MDS_SemTake(FLASH 信号量);
00008
00009          MDS_SemGive(FLASH 信号量);
00010      }
00011
00012      MDS_SemGive(FLASH 信号量);
00013  }
```

在 00001 行，FlashErase 函数先获取到了 FLASH 信号量，运行到 00007 行时又需要再次获取 FLASH 信号量，按照二进制信号量的特性，那么在 00007 行时将无法获取到 FLASH 信号量，任务被自己阻塞了。

互斥信号量可以解决这个问题，互斥信号量是以任务为管理单元的，只要一个任务一旦获取到互斥信号量之后，那么这个任务就可以再次重复获取到该信号量，而其它任务则无法获取到该信号量。当然，获取到互斥信号量的任务需要释放该信号量时，它需要保证释放该互斥信号量的次数和获取该互斥信号量的操作次数是等量的，也就是说同一个任务获

取几次互斥信号量，也必须释放几次互斥信号量，这样才能彻底释放互斥信号量，不然会其它任务将无法获取到这个互斥信号量。

互斥信号量的这个特性是与任务相关的，因此我们需要在 SEM 结构体中增加一个任务指针变量 pstrSemTask，用它来关联获取到互斥信号量的任务。

```
typedef struct m_sem
{
    M_TASKSCHEDTAB strSemTab;       /* 信号量调度表 */
    U32 uiCounter;                  /* 信号量计数值 */
    U32 uiSemOpt;                   /* 信号量参数 */
    struct m_tcb* pstrSemTask;      /* 获取到互斥信号量的任务 */
}M_SEM;
```

互斥信号量对外仍表现出空满两种状态，但对内仍需要记录同一任务获取、释放信号量的次数，我们仍可以像计数信号量那样使用 uiCounter 变量来实现这个功能。互斥信号量在创建时必须为满状态，保证第一个获取该信号量的任务可以获取到该信号量。pstrSemTask 指针在初始化时指向 NULL，表明互斥信号量还没有被任何任务获取，当一个任务获取到该互斥信号量时，需要将 pstrSemTask 指针修改为任务自己的 TCB 指针，表明该互斥信号量已经被该任务获取，当该任务再次获取该信号量时，发现 pstrSemTask 指针记录的是自己的 TCB 指针，那么只将 uiCounter 变量做自减计数。当其它函数获取该互斥信号量时，发现 pstrSemTask 指针并不是自己的 TCB 指针，则直接返回，不能获取到该互斥信号量。当任务释放互斥信号量时，也需要检查互斥信号量的 pstrSemTask 指针，若是自己的 TCB 指针则将 uiCounter 变量自加，当处于信号量处于满状态时则完全释放信号量。如果 pstrSemTask 指针不是自己的 TCB 指针则直接返回失败。

上面介绍了互斥信号量的原理，下面将对信号量相关函数中有关互斥信号量的代码做一下简单的介绍：

```
00024  U32 MDS_SemCreate(M_SEM* pstrSem, U32 uiSemOpt, U32 uiInitVal)
00025  {
00026
.....      .....
00031
00032      /* 信号量选项检查 */
00033      if(((SEMBIN != (SEMTYPEMASK & uiSemOpt))
00034          && (SEMCNT != (SEMTYPEMASK & uiSemOpt))
00035          && (SEMMUT != (SEMTYPEMASK & uiSemOpt)))
00036         || ((SEMFIFO != (SEMSCHEDOPTMASK & uiSemOpt))
00037             && (SEMPRIO != (SEMSCHEDOPTMASK & uiSemOpt))))
00038      {
00039          return RTN_FAIL;
00040      }
00041
.....      .....
00049
00050      /* 互斥信号量初始值只能是满 */
00051      else if(SEMMUT == (SEMTYPEMASK & uiSemOpt))
00052      {
00053          if(SEMFULL != uiInitVal)
00054          {
00055              return RTN_FAIL;
00056          }
00057      }
00058
.....      .....
```

```
00067
00068       /* 没有任务获取到互斥信号量 */
00069       pstrSem->pstrSemTask = (M_TCB*)NULL;
00070
00071       return RTN_SUCD;
00072   }
```

00033~00040 行，对入口参数进行检查，只能创建二进制信号量、计数信号量和互斥信号量，只能创建 FIFO 或者 PRIO 类型的信号量。

00051~00057 行，只能创建满状态的互斥信号量。

00069 行，新创建的信号量的 pstrSemTask 指针需要初始化为 NULL。

```
00091   U32 MDS_SemTake(M_SEM* pstrSem, U32 uiDelayTick)
00092   {
00093
……          ……
00103
00104       /* 获取信号量时不等待时间 */
00105       if(SEMNOWAIT == uiDelayTick)
00106       {
00107
……              ……
00146
00147           else /* 互斥信号量 */
00148           {
00149               /* 信号量为满，可获取到信号量 */
00150               if(SEMFULL == pstrSem->uiCounter)
00151               {
00152                   /* 获取到信号量后将信号量计数值-1 */
00153                   pstrSem->uiCounter--;
00154
00155                   /* 将互斥信号量与任务关联 */
00156                   pstrSem->pstrSemTask = gpstrCurTcb;
00157
00158                   (void)MDS_IntUnlock();
00159
00160                   return RTN_SUCD;
00161               }
00162               else /* 信号量不为满，说明被一个任务获取过 */
00163               {
00164                   /* 若该信号量已经被本任务获取则可以继续获取 */
00165                   if(pstrSem->pstrSemTask == gpstrCurTcb)
00166                   {
00167                       /* 信号量计数未空 */
00168                       if(SEMEMPTY != pstrSem->uiCounter)
00169                       {
00170                           /* 信号量计数值-1 */
00171                           pstrSem->uiCounter--;
00172
00173                           (void)MDS_IntUnlock();
00174
00175                           return RTN_SUCD;
00176                       }
00177                       else /* 信号量计数已空，返回失败 */
00178                       {
00179                           (void)MDS_IntUnlock();
00180
```

```
00181                           return RTN_SMTKOV;
00182                       }
00183                   }
00184                   else /* 被其他任务获取则返回无法获取信号量 */
00185                   {
00186                       (void)MDS_IntUnlock();
00187
00188                       return RTN_SMTKRT;
00189                   }
00190               }
00191           }
00192
......            ......
00191
00192       }
00193       else /* 获取信号量时需要等待时间 */
00194       {
00195
......            ......
00260
00261           else /* 互斥信号量 */
00262           {
00263               /* 信号量为满, 可获取到信号量 */
00264               if(SEMFULL == pstrSem->uiCounter)
00265               {
00266                   /* 获取到信号量后将信号量计数值-1 */
00267                   pstrSem->uiCounter--;
00268
00269                   /* 将互斥信号量与任务关联 */
00270                   pstrSem->pstrSemTask = gpstrCurTcb;
00271
00272                   (void)MDS_IntUnlock();
00273
00274                   return RTN_SUCD;
00275               }
00276               else /* 信号量不为满, 说明被一个任务获取过 */
00277               {
00278                   /* 若该信号量已经被本任务获取则可以继续获取 */
00279                   if(pstrSem->pstrSemTask == gpstrCurTcb)
00280                   {
00281                       /* 信号量计数未空 */
00282                       if(SEMEMPTY != pstrSem->uiCounter)
00283                       {
00284                           /* 信号量计数值-1 */
00285                           pstrSem->uiCounter--;
00286
00287                           (void)MDS_IntUnlock();
00288
00289                           return RTN_SUCD;
00290                       }
00291                       else /* 信号量计数已空, 返回失败 */
00292                       {
00293                           (void)MDS_IntUnlock();
00294
00295                           return RTN_SMTKOV;
00296                       }
00297                   }
00298                   else /* 被其他任务获取, 需要切换任务 */
00299                   {
```

```
00300                       /* 将任务置为 pend 状态 */
00301                       if(RTN_FAIL == MDS_TaskPend(pstrSem, uiDelayTick))
00302                       {
00303                           (void)MDS_IntUnlock();
00304
00305                           /* 任务 pend 失败 */
00306                           return RTN_FAIL;
00307                       }
00308
00309                       (void)MDS_IntUnlock();
00310
00311                       /* 使用软中断调度任务 */
00312                       MDS_TaskSwiSched();
00313
00314                       /* 任务pend的返回值, 该值在任务pend状态结束时被保存在uiDelayTick
00315                          中 */
00316                       return gpstrCurTcb->strTaskOpt.uiDelayTick;
00317                   }
00318               }
00319           }
00320       }
00321   }
```

00150~00161 行，互斥信号量为满状态，说明此次为任务第一次获取该互斥信号量，信号量计数 uiCounter 自减，并将该任务的 TCB 关联到信号量的 pstrSemTask 指针变量上。

00171 行，走到此分支说明该信号量已经被该任务获取，在信号量计数还没有达到空状态的情况下计数 uiCounter 自减。

00181 行，信号量计数已经为空了，不能再自减了，返回失败。

00188 行，走到此行说明该互斥信号量已经被其它任务获取，当前任务无法获取，在不需要等待的情况下直接返回失败。

00264~00297 行与 00150~00182 行的执行过程是相同的。

00298 行，走此分支说明该互斥信号量已经被其它任务获取，当前任务无法获取到该信号量，该获取信号量的动作需要等待时间。

00301~00307 行，将任务阻塞到互斥信号量上。

00312 行，信号量操作完毕后，调用软中断调度函数，重新调度任务。

00316 行，返回任务的返回值。

```
00330   U32 MDS_SemGive(M_SEM* pstrSem)
00331   {
00332
......      ......
00386
00387       else /* 互斥信号量 */
00388       {
00389           /* 若释放互斥信号量的任务不是获取到互斥信号量的任务, 则返回失败 */
00390           if(pstrSem->pstrSemTask != gpstrCurTcb)
00391           {
00392               (void)MDS_IntUnlock();
00393
00394               return RTN_FAIL;
00395           }
00396
00397           /* 释放一次互斥信号量, 信号量计数值+1 */
00398           pstrSem->uiCounter++;
```

```
00399
00400           /* 同一任务获取和释放互斥信号量不平衡，不需要检查被该信号量挂起的任务 */
00401           if(SEMFULL != pstrSem->uiCounter)
00402           {
00403               ucPendTaskFlag = 0;
00404           }
00405           else /* 同一任务获取和释放互斥信号量平衡，需要检查被该信号量挂起的任务 */
00406           {
00407               ucPendTaskFlag = 1;
00408           }
00409       }
00410
……                  ……
00449
00450               /* 为所激活的被互斥信号量阻塞的任务赋初值 */
00451               if(SEMMUT == (SEMTYPEMASK & pstrSem->uiSemOpt))
00452               {
00453                   pstrSem->pstrSemTask = pstrTcb;
00454                   pstrSem->uiCounter--;
00455               }
00456
……                  ……
00477
00478               else /* 互斥信号量 */
00479               {
00480                   /* 没有任务获取到互斥信号量 */
00481                   pstrSem->pstrSemTask = (M_TCB*)NULL;
00482               }
00483
……                  ……
00488
00489   }
```

00390~00395 行，互斥信号量已经被其它任务获取，当前任务无法释放，返回失败。

00398 行，走到此处说明互斥信号量已经被当前任务获取，计数变量 uiCounter 需要自加 1。

00401~00404 行，信号量若不处于满状态，说明当前任务还没有将互斥信号量释放完毕，不需要检查被该信号量挂起的任务。

00405~00408 行，信号量已经处于满状态，说明当前任务已经将互斥信号量释放完毕，需要检查被该信号量挂起的任务。

00451~00455 行，走到此分支说明互斥信号量已经被一个任务释放完毕，并且有另外一个任务被阻塞在这个信号量上，该被阻塞的任务需要被激活并获取到这个信号量。将这个被激活任务的 TCB 关联到这个信号量上，由于信号量重新被获取，需要将 uiCounter 变量自减 1。

00478~00482 行，走到此分支说明互斥信号量已经被一个任务释放完毕，并且没有任务被阻塞在这个信号量上，直接将 NULL 关联这个信号量上。

上面就是本节新增加的互斥信号量的功能，互斥信号量只能由获取到它的任务获取，也只能由获取到它的任务释放，除此之外，互斥信号量还有其它特性，比如任务删除保护、任务优先级继承等特性，这些特性我们将在后面章节再去实现。

使用互斥信号量也有一些限制，互斥信号量只能用于任务间的互斥，不能用于同步，互斥信号量不能在中断中使用，不能对互斥信号量使用 MDS_SemFlush 函数。

本节我们使用 2 个任务 TEST_TestTask1 和 TEST_TestTask2 来验证互斥信号量的功能，并设计 3 个函数，TEST_Test1、TEST_Test2 和 TEST_Test3，这三个函数都使用同一个互斥信号量，开始时获取信号量，延迟一段时间后再释放信号量，周而复始的如此运行。其中 TEST_Test1 函数在信号量保护期间会调用 TEST_Test2 函数，TEST_TestTask1 任务周而复始的执行 TEST_Test1 函数，TEST_TestTask2 任务周而复始的执行 TEST_Test3 函数，当 TEST_TestTask1 任务运行时，我们应该能看到 TEST_Test1 和 TEST_Test2 函数的打印输出，而当 TEST_TestTask2 任务运行时，我们应该只能看到 TEST_Test3 函数的输出。

```
00018  void TEST_TestTask1(void)
00019  {
00020      while(1)
00021      {
00022          DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00023                          MDS_SystemTickGet());
00024
00025          TEST_Test1();
00026      }
00027  }

00034  void TEST_TestTask2(void)
00035  {
00036      while(1)
00037      {
00038          DEV_PutStrToMem((U8*)"\r\nTask2 is running! Tick is: %d",
00039                          MDS_SystemTickGet());
00040
00041          TEST_Test3();
00042      }
00043  }

00050  void TEST_Test1(void)
00051  {
00052      /* 获取到信号量才运行 */
00053      (void)MDS_SemTake(&gstrSemMut, SEMWAITFEV);
00054
00055      DEV_PutStrToMem((U8*)"\r\nT1 is running! Tick is: %d", MDS_SystemTickGet());
00056
00057      DEV_DelayMs(1500);
00058
00059      TEST_Test2();
00060
00061      (void)MDS_TaskDelay(100);
00062
00063      (void)MDS_SemGive(&gstrSemMut);
00064
00065      (void)MDS_TaskDelay(100);
00066  }

00073  void TEST_Test2(void)
00074  {
00075      /* 获取到信号量才运行 */
00076      (void)MDS_SemTake(&gstrSemMut, SEMWAITFEV);
00077
00078      DEV_PutStrToMem((U8*)"\r\nT2 is running! Tick is: %d", MDS_SystemTickGet());
00079
00080      DEV_DelayMs(500);
00081
00082      (void)MDS_TaskDelay(200);
```

```
00083
00084       (void)MDS_SemGive(&gstrSemMut);
00085
00086       (void)MDS_TaskDelay(300);
00087   }

00094   void TEST_Test3(void)
00095   {
00096       /* 获取到信号量才运行 */
00097       (void)MDS_SemTake(&gstrSemMut, SEMWAITFEV);
00098
00099       DEV_PutStrToMem((U8*)"\r\nT3 is running! Tick is: %d", MDS_SystemTickGet());
00100
00101       DEV_DelayMs(500);
00102
00103       (void)MDS_TaskDelay(200);
00104
00105       (void)MDS_SemGive(&gstrSemMut);
00106
00107       (void)MDS_TaskDelay(200);
00108   }
```

编译本节代码，运行输出的数据截图如下：

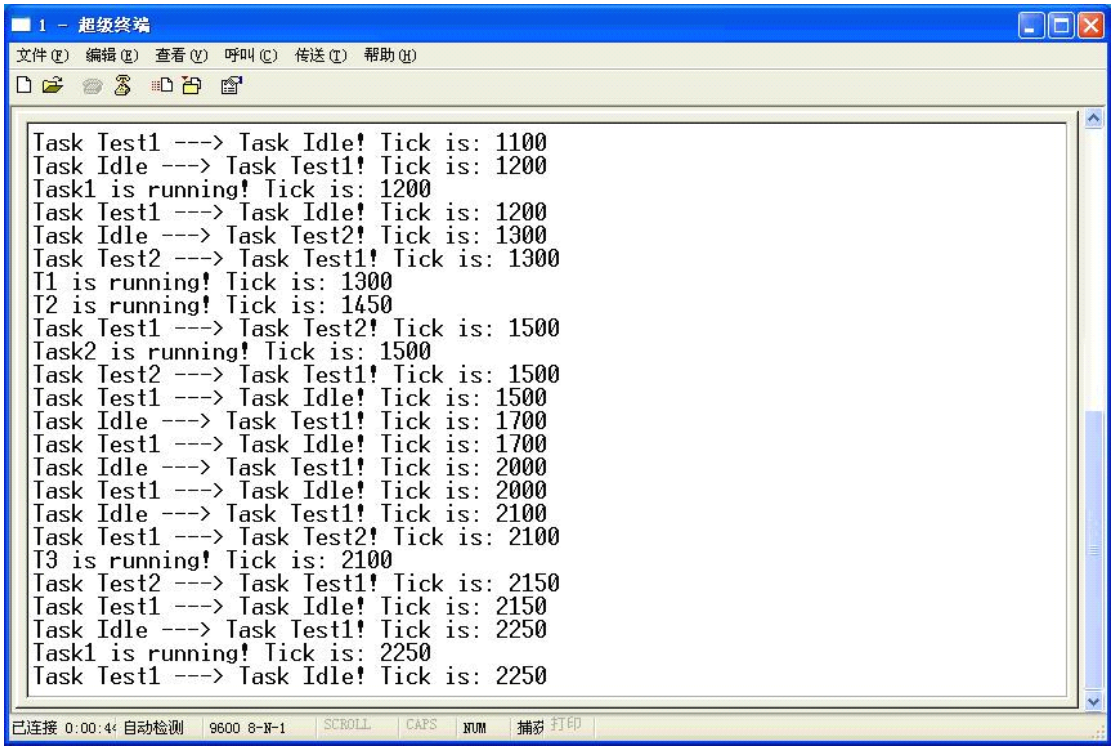


图 59　互斥信号量的打印信息

读者可以登录 http://blog.sina.com.cn/ifreecoding 网站下载视频观看动态输出过程。

使用工具软件解析本节任务切换过程的数据，结果如下图所示：

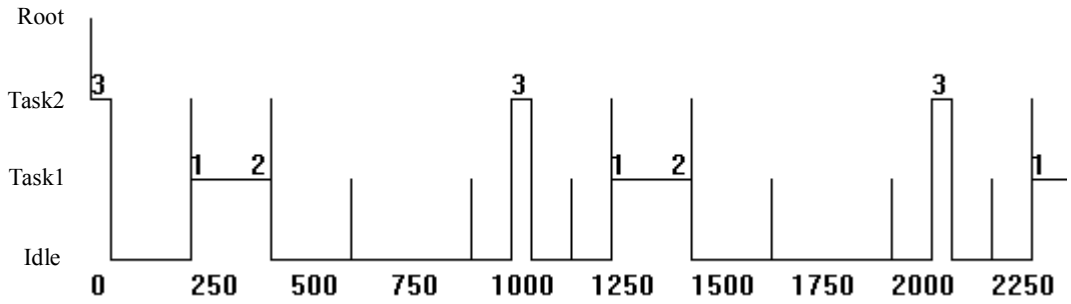


图 60　4.9 节任务切换过程图

从图 60 可以看到同一任务 TEST_TestTask1 中的两个函数 TEST_Test1 和 TEST_Test2 可以重复获取到互斥信号量，而不同的两个任务 TEST_TestTask1 和 TEST_TestTask2 之间由互斥信号量实现了任务的互斥运行。

第 10 节 队列

本节将完善队列功能，使队列也具有触发系统任务调度的功能，用户可以使用队列来传递消息、同步事件。至此，Mindows 操作系统将提供操作系统最基本的任务调度功能、信号量功能和队列功能。

队列正如其名，可以将消息按照顺序排放在队列里，如同排队一样，先来的排在前面后来的排在后面，从队列中取消息时也是先从队列头部开始取，当队列为空时，取消息的任务会被阻塞，发生任务切换。

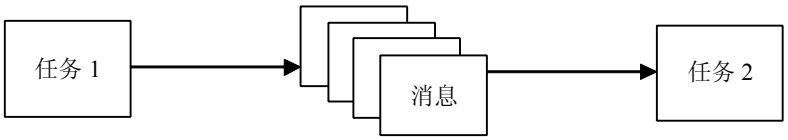


图 61　任务间使用队列传递消息

队列是一个传递消息的通道，可以有 1 进 1 出或多进 1 出等多种形式，发送端只需要将消息压入队列，接收端只需要从队列中获取消息，通过队列这种方式可以在多个任务间方便的实现各种消息的传递。

使用 MDS_QuePut 函数可以将消息存入队列，使用 MDS_QueGet 函数可以从队列中获取消息，队列中可以同时存在多个消息，按照先进先出的顺序存放，当队列为空时调用 MDS_QueGet 函数的任务将被阻塞。上述这些描述与计数信号量非常相似，计数信号量使用 MDS_SemGive 函数将信号量计数值存入信号量，使用 MDS_SemTake 函数从信号量中获取信号量计数值，计数信号量中可以保存信号量计数值，当计数信号量为空时调用 MDS_SemTake 函数的任务将被阻塞。

我们可以将计数信号量封装到队列内部，使用计数信号量实现队列阻塞、唤醒任务的功能，在原有的队列结构中需要增加一个信号量的结构，如下：

```
typedef struct m_que     /* 队列结构 */
```

```
{
    M_CHAIN strChain;   /* 队列链表 */
    M_SEM strSem;       /* 队列信号量 */
}M_QUE;
```

创建队列时将队列结构中的信号量结构创建为空的计数信号量，将队列链表初始化为空。存入到队列的消息需要有一个 M_CHAIN 的链表结构，存入队列时将此 M_CHAIN 链表结构挂接到队列结构的 strChain 根链表上，并释放一次队列的 strSem 计数信号量，使信号量的计数值与队列中的消息数保持一致。当从队列中获取消息时首先获取队列中的 strSem 计数信号量，若信号量为空则说明队列也为空，任务就会被信号量阻塞，实现了队列为空阻塞任务的功能。若信号量不为空 strSem 信号量的计数值会减 1，并从队列的 strChain 链表中获取一个节点，取出队列中的消息。删除队列时需要使用 MDS_SemFlushValue 函数释放被队列阻塞的所有任务。

上述是实现队列的原理，代码比较简单，不再详细介绍，仅列出代码如下：

```
00017  U32 MDS_QueCreate(M_QUE* pstrQue, U32 uiSemOpt)
00018  {
00019      /* 入口参数检查 */
00020      if(NULL == pstrQue)
00021      {
00022          return RTN_FAIL;
00023      }
00024
00025      /* 队列选项检查 */
00026      if((QUEFIFO != (QUESCHEDOPTMASK & uiSemOpt))
00027         && (QUEPRIO != (QUESCHEDOPTMASK & uiSemOpt)))
00028      {
00029          return RTN_FAIL;
00030      }
00031
00032      MDS_ChainInit(&pstrQue->strChain);
00033
00034      /* 创建队列使用的计数信号量 */
00035      if(RTN_SUCD == MDS_SemCreate(&pstrQue->strSem, SEMCNT | uiSemOpt, SEMEMPTY))
00036      {
00037          return RTN_SUCD;
00038      }
00039      else
00040      {
00041          return RTN_FAIL;
00042      }
00043  }
```

可以选择创建 FIFO 或 PRIO 属性的队列，这个属性是针对被队列阻塞的任务的，而不是放入队列中的消息。在 Mindows 中，队列中的消息都是按照 FIFO 形式存放的，当然，如果你愿意的话也可以自己增加新函数，实现 PRIO 形式存放的队列。

```
00053  U32 MDS_QuePut(M_QUE* pstrQue, M_CHAIN* pstrQueNode)
00054  {
00055      /* 入口参数检查 */
00056      if((NULL == pstrQue) || (NULL == pstrQueNode))
00057      {
00058          return RTN_FAIL;
00059      }
00060
```

```
00061     (void)MDS_IntLock();
00062
00063     /* 将节点加入队列 */
00064     MDS_ChainNodeAdd(&pstrQue->strChain, pstrQueNode);
00065
00066     (void)MDS_IntUnlock();
00067
00068     return MDS_SemGive(&pstrQue->strSem);
00069 }

00086 U32 MDS_QueGet(M_QUE* pstrQue, M_CHAIN** ppstrQueNode, U32 uiDelayTick)
00087 {
00088     M_CHAIN* pstrQueNode;
00089     U32 uiRtn;
00090
00091     /* 入口参数检查 */
00092     if((NULL == pstrQue) || (NULL == ppstrQueNode))
00093     {
00094         return RTN_FAIL;
00095     }
00096
00097     /* 没有获取到队列需要的信号量，返回失败 */
00098     uiRtn = MDS_SemTake(&pstrQue->strSem, uiDelayTick);
00099     if(RTN_SUCD != uiRtn)
00100     {
00101         return uiRtn;
00102     }
00103
00104     (void)MDS_IntLock();
00105
00106     /* 从队列取出节点 */
00107     pstrQueNode = MDS_ChainNodeDelete(&pstrQue->strChain);
00108
00109     (void)MDS_IntUnlock();
00110
00111     /* 取出节点 */
00112     *ppstrQueNode = pstrQueNode;
00113
00114     return RTN_SUCD;
00115 }

00123 U32 MDS_QueDelete(M_QUE* pstrQue)
00124 {
00125     /* 入口参数检查 */
00126     if(NULL == pstrQue)
00127     {
00128         return RTN_FAIL;
00129     }
00130
00131     /* 释放队列信号量所阻塞的所有任务，返回信号量被删除 */
00132     if(RTN_SUCD != MDS_SemFlushValue(&pstrQue->strSem, RTN_SMTKDL))
00133     {
00134         return RTN_FAIL;
00135     }
00136
00137     return RTN_SUCD;
00138 }
```

本节的消息打印功能用队列来实现，TEST_TestTask1 和 TEST_TestTask2 任务将打印信息打印到消息缓冲中，这些消息缓冲被压入队列，idle 任务则查询队列，如果队列不为空则

说明有消息需要打印，从队列中取出这些消息缓冲打印到串口，如果队列为空 idle 任务则取不到消息，由于 idle 任务不能被阻塞，因此在获取队列消息时使用的是 QUENOWAIT 参数，获取不到消息时直接退出本次循环，直到其它任务将消息放入队列中，idle 任务再打印出新消息，这样就实现了打印消息的功能。

```
00016  void TEST_TestTask1(void)
00017  {
00018      while(1)
00019      {
00020          DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00021                          MDS_SystemTickGet());
00022
00023          DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00024                          MDS_SystemTickGet());
00025
00026          DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00027                          MDS_SystemTickGet());
00028
00029          DEV_DelayMs(2000);
00030
00031          (void)MDS_TaskDelay(300);
00032      }
00033  }
```

```
00040  void TEST_TestTask2(void)
00041  {
00042      while(1)
00043      {
00044          DEV_PutStrToMem((U8*)"\r\nTask2 is running! Tick is: %d",
00045                          MDS_SystemTickGet());
00046
00047          DEV_PutStrToMem((U8*)"\r\nTask2 is running! Tick is: %d",
00048                          MDS_SystemTickGet());
00049
00050          DEV_DelayMs(500);
00051
00052          (void)MDS_TaskDelay(200);
00053      }
00054  }
```

编译本节代码，运行输出截图如下：

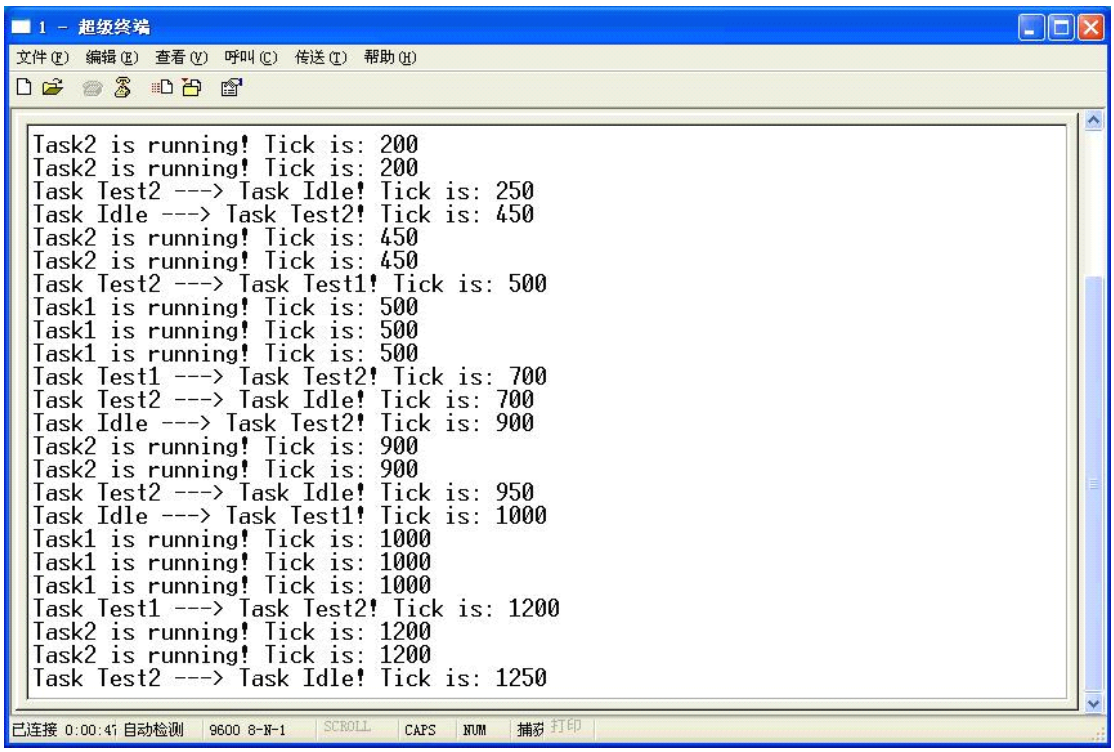


图 62　使用队列打印消息

打印消息连续输出的过程可登录 http://blog.sina.com.cn/ifreecoding 网站下载视频观看。使用工具软件解析本节任务切换过程的数据，结果如下图所示：

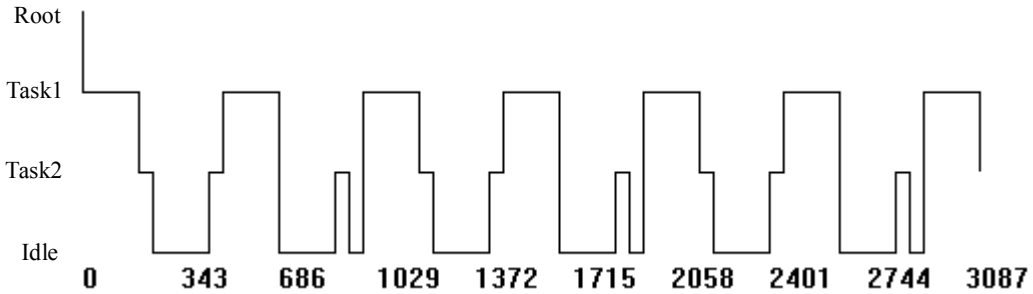


图 63　4.10 节任务切换过程图

第 5 章 将操作系统移植到 Cortex 内核的芯片上

本章我们将会把 Wanlix 和 Mindows 移植到 cortex 内核的芯片上，通过这个过程为大家展示一下代码移植的方法，加深大家对这 2 个操作系统的理解，之后我们将在 cortex 内核的芯片上继续完善 Mindows 的功能。

我选取了 2 个 cortex 内核的芯片，一个是 TI 的 LM3S8962 芯片，另一个是 ST 的 STM32F103VB 芯片。

第 1 节 Cortex 内核介绍

LM3S8962 和 STM32F103VB 芯片都是采用 Cortex - M3 的内核，Cortex 内核是 ARM 公司推出的最新款处理器内核，其中 M3 系列主要面向低端市场，Cortex - M3 内核处理器强大的功能和超高的性价比在某些领域对传统的单片机形成了严峻的挑战。

前面所使用的 Aduc7024 芯片采用的是 ARM7TDMI 内核，是 ARMv4T 的架构，Cortex - M3 内核采用的是 ARMv7 架构。Cortex - M3 是一个 32 位的内核，内部总线是 32 位的，寄存器是 32 位的，但它所采用的 Thumb-2 指令集却实现了 16/32bits 指令混合使用，同时实现了高代码密度和高执行速度，而无需像以前的 ARM 处理器为了在代码密度与执行速度之间寻求一个平衡而需要在 ARM 指令集与 Thumb 指令集之间切换。

下面 3 段汇编程序是我将同一段 C 语言代码分别编译成 ARM、Thumb 和 Thumb-2 指令而得到的：

◆ ARM 指令集：

```
指令地址        机器码          汇编语言
800e4:          e24dd050        sub     sp, sp, #80     ; 0x50
800e8:          e3a00000        mov     r0, #0
800ec:          e58d004c        str     r0, [sp, #76]     ; 0x4c
800f0:          ea000005        b       8010c <main+0x28>
800f4:          e3a00000        mov     r0, #0
800f8:          e59d104c        ldr     r1, [sp, #76]     ; 0x4c
800fc:          e78d0101        str     r0, [sp, r1, lsl #2]
80100:          e59d004c        ldr     r0, [sp, #76]     ; 0x4c
80104:          e2800001        add     r0, r0, #1
80108:          e58d004c        str     r0, [sp, #76]     ; 0x4c
8010c:          e59d004c        ldr     r0, [sp, #76]     ; 0x4c
80110:          e3500014        cmp     r0, #20
80114:          3afffff6        bcc     800f4 <main+0x10>
80118:          e3a00000        mov     r0, #0
8011c:          e28dd050        add     sp, sp, #80     ; 0x50
80120:          e12fff1e        bx      lr
```

◆ Thumb 指令集：

```
指令地址        机器码          汇编语言
800e4:          b094            sub     sp, #80     ; 0x50
800e6:          2000            movs    r0, #0
```

```
800e8:      9013            str     r0, [sp, #76]    ; 0x4c
800ea:      e007            b.n     800fc <main+0x18>
800ec:      2000            movs    r0, #0
800ee:      9913            ldr     r1, [sp, #76]    ; 0x4c
800f0:      0089            lsls    r1, r1, #2
800f2:      466a            mov     r2, sp
800f4:      5050            str     r0, [r2, r1]
800f6:      9813            ldr     r0, [sp, #76]    ; 0x4c
800f8:      1c40            adds    r0, r0, #1
800fa:      9013            str     r0, [sp, #76]    ; 0x4c
800fc:      9813            ldr     r0, [sp, #76]    ; 0x4c
800fe:      2814            cmp     r0, #20
80100:      d3f4            bcc.n   800ec <main+0x8>
80102:      2000            movs    r0, #0
80104:      b014            add     sp, #80    ; 0x50
80106:      4770            bx      lr
```

◆ Thumb-2 指令集:

```
指令地址    机器码          汇编语言
80100:      b094            sub     sp, #80    ; 0x50
80102:      f04f 0000       mov.w   r0, #0
80106:      9013            str     r0, [sp, #76]    ; 0x4c
80108:      e008            b.n     8011c <main+0x1c>
8010a:      f04f 0000       mov.w   r0, #0
8010e:      9913            ldr     r1, [sp, #76]    ; 0x4c
80110:      f84d 0021       str.w   r0, [sp, r1, lsl #2]
80114:      9813            ldr     r0, [sp, #76]    ; 0x4c
80116:      f100 0001       add.w   r0, r0, #1
8011a:      9013            str     r0, [sp, #76]    ; 0x4c
8011c:      9813            ldr     r0, [sp, #76]    ; 0x4c
8011e:      2814            cmp     r0, #20
80120:      d3f3            bcc.n   8010a <main+0xa>
80122:      2000            movs    r0, #0
80124:      b014            add     sp, #80    ; 0x50
80126:      4770            bx      lr
```

从上面这 3 段功能相同的汇编代码可以大致看出 ARM、Thumb 和 Thumb-2 指令集的特点，我们整理一下形成下表：

	指令数	指令空间	指令特点
ARM 指令集	16 条	64 字节	全部指令都是 32bits
Thumb 指令集	18 条	36 字节	全部指令都是 16bits
Thumb-2 指令集	16 条	40 字节	16/32bits 指令混合组成

表 11 ARM、Thumb 和 Thumb-2 指令对比

其中 ARM 指令集的汇编代码全部是 32bits 的，每条指令能承载更多的信息，因此它使用了最少的指令完成了功能，在相同频率下运行速度也是最快的，但也因为每条指令是最长的而占用了最多的程序空间。Thumb 指令集的汇编代码全部是 16bits 的，每条指令所能承载的信息少，因此它需要使用更多的指令才能完成功能，因此运行速度慢，但它也占用了最少的程序空间。而 Thumb-2 指令集则在这两者之间取了一个平衡，兼有二者的优势，当一个操作可以使用一条 32bits 指令完成时就使用 32bits 的指令，加快运行速度，而当一次操作只需要一条 16bits 指令完成时就使用 16bits 的指令，节约存储空间。

上面这 3 段汇编程序只是一小段 C 程序编译后的结果，样本数太小，只能大概说明情况。

除了指令集变化之外，cortex 内核的工作模式也大为简化，只有 2 种模式：handler 模式和 thread 模式，与之对应，cortex 内核的操作权限分为了 2 个等级：特权级和用户级，特权级可以访问芯片内部的一切资源，对芯片具有完全的控制权，而用户级则无法访问芯片内部的一些重要寄存器，只能使用程序运行所需的基本资源。handler 模式运行在特权级下，而 thread 模式既可以运行在特权级下也可以运行在用户级下。

芯片复位后处理器会自动进入 thread 模式的特权级，这时程序具有控制芯片的全部权利，可以对芯片进行配置。当芯片配置完毕，程序应该主动切换到 thread 模式的用户级，放弃控制芯片中重要寄存器的权利，以防止程序意外甚至是恶意的破坏芯片中那些已配置好的寄存器。如果用户级程序出现错误，超出了用户级的访问权限，芯片就会产生异常，自动切换到 handler 模式，进入中断向量表中相应的异常服务程序，此时程序具有特权级，可以对出现的异常进行处理。

可以通过设置 CONTROL 寄存器的 bit 0 位从特权级切换到用户级，而从用户级切换到特权级则比较麻烦，必须通过异常中断才能返回到特权级。如果想在退出异常中断后还保留 thread 模式的特权级，那么就必须在异常中断所处的特权级情况下修改 CONTROL 寄存器的 bit 0 来实现。总之，从特权级切换到用户级很简单，而从用户级切换到特权级则必须通过触发异常中断来实现，这个异常中断就好比是一个审批部门，掌握着是否给某些程序行使特权的权利，它又好比是一个监督部门，可以在中断中对出现的异常进行处理。

利用这 2 种特权级可以保护程序更安全的运行，在需要完全控制芯片时使用特权级，而给一般的应用程序只开放用户级权限，这样就算用户级程序出现了异常也未必会对芯片造成致命的影响。

随着芯片工作模式数量的减少，与之对应的内部的寄存器数量也减少了，只保留了 13 个通用寄存器 R0~R12，2 个堆栈指针寄存器 SP，1 个链接寄存器 LR，1 个程序指针寄存器 PC，还有 1 个状态寄存器 XPSR。

R0~R12 这 13 个通用寄存器用法没什么变化，可以参考第 2 章的介绍。

2 个堆栈指针寄存器分别是 MSP 和 PSP，MSP 是芯片缺省使用的堆栈指针。handler 模式下只能使用 MSP 指针，thread 模式下可以使用 MSP 或 PSP 指针，至于使用哪个指针则需要在 thread 模式的特权级下对 CONTROL 寄存器的 bit 1 位进行配置。注意，在程序运行的任何时刻我们只能看到 MSP 或 PSP，这两个堆栈寄存器不能存在，这点与 ARM7 中 USR 模式下的 SP 不能和其它模式下的 SP 同时存在是一样的。

如果你不希望把程序搞得太复杂，你完全可以只使用 MSP 这一个堆栈指针，当然同时使用 MSP 和 PSP 也是有好处的，比如我们可以在用户级的程序中使用 PSP 指针，而在异常中断中使用 MSP 指针，这样当用户程序的 PSP 指针被不小心破坏的时候，程序必然会出现错误，必然会引发异常中断，当异常中断发生时，芯片自动切换到了 handler 模式，权限也随之切换到了特权级，堆栈指针也随之切换到了 MSP，这样芯片就可以在异常中断中处理这个错误了。如果只使用一个 MSP 指针的话，尽管程序进入了异常中断，但 MSP 指针已经在用户模式下被破坏了，那么这个异常中断也会因为这个无效的指针而无法运行，整个芯片就挂死了。

LR 寄存器的功能在一般的函数调用时没有发生变化，还是用来存储调用函数后的返回地址，但 cortex 内核在中断发生时对 LR 的处理方式与 ARM7 内核却有所不同，cortex 内核在中断发生时，硬件会自动的将 XPSR、PC、LR、R12、R3、R2、R1 和 R0 压入栈，如果当前在使用 MSP 则将这 8 个寄存器压入 MSP 栈，如果当前在使用 PSP 则将这 8 个寄存器压入 PSP 栈，但进入中断后就一定是使用 MSP 了。进到中断服务程序里，硬件会自动将一个特殊的值“EXC_RETURN”存入 LR 寄存器中，EXC_RETURN 这个值只能是 0xFFFFFFF1、0xFFFFFFF9 或者是 0xFFFFFFFD 这 3 个值中的一个，0xFFFFFFF1 代表中断服务程序将返回到 handler 模式，

使用 MSP，0xFFFFFFF9 代表中断服务程序将返回到 handler 模式，使用 MSP，0xFFFFFFFD 代表中断服务程序将返回到 handler 模式，使用 PSP。在中断服务程序返回时，必须跳转到 EXC_RETURN 这个值，而不能直接跳转到中断服务程序返回时的地址，中断服务程序返回后硬件将根据 EXC_RETURN 值的定义使用对应的模式和堆栈寄存器。这种方式是设置 CONTROL 寄存器的 bit 1 位之外的另一种选择 MSP 或 PSP 的方式。

既然 LR 中没有保存返回地址，那么处理器怎么知道中断结束后应该从哪个地址继续运行？别忘了在进入中断前，硬件自动将 8 个寄存器压入了堆栈，这其中包含有 PC 寄存器，压入 PC 的值就是中断发生时的下条指令所在的地址，也就是中断返回的地址。在中断返回时，硬件还会自动从堆栈中恢复这 8 个寄存器的内容，这样程序就可以在中断返回后继续运行了。

PC 寄存器的功能没有变，保存的是指令的地址。Thumb-2 指令是由 16/32bits 指令混合组成的，指令需要 2/4 字节对齐，因此指令所在地址的最低 bit 也就是 PC 寄存器的 bit 0 需要保持为 0，在读取 PC 寄存器时确实如此,但 Thumb-2 指令集与 Thumb 指令集一样，规定在写 PC 寄存器的时候 PC 寄存器的 bit 0 必须为 1，以表明当前正在使用 Thumb-2 指令集而不是 ARM 指令集（ARM 指令集为 0），这个 bit 0 中的 1 不是说指令地址是奇地址对齐，而是指明了当前使用的指令集。

Cortex 内核只保留了一个状态寄存器——XPSR，这个寄存器是由多个寄存器复合而成的，如下图所示：

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR												Exception number				
EPSR						ICI/IT	T			ICI/IT						

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

图 64　XPSR 寄存器

其中APSR寄存器中存放着应用程序的状态，与ARM7内核的CPSR寄存器中的NZCVQ功能一样。IPSR寄存器中保存着中断号，比如发生了中断14，那么该寄存器的值就为14。EPSR寄存器中保存着执行程序的状态，其中T标志就是表明当前指令是否是Thumb-2指令，ICI/IT指明某些指令的运行状态。

APSR、IPSR、EPSR这3个寄存器组合在一起就形成了XPSR寄存器，我们可以使用MSR/MRS指令访问这些寄存器，即可以使用这3个寄存器的名字单独访问，也可以使用XPSR这个名字对它们一起访问。

Cortex内核支持255个中断，但芯片厂商在设计芯片时一般并不会支持这么多的中断，这些中断绝大多数是可以修改优先级的，高优先级的中断可以抢占正在执行的低优先级中断，由此，cortex内核实现了“晚到中断”和“咬尾中断”机制，所谓晚到中断是指在低优先级中断压栈的过程中又发生了高优先级的中断，那么这个压栈过程就算是这个高优先级中断的压栈，压栈之后执行高优先级中断。所谓咬尾中断是指高优先级中断服务程序在执行过程中发生了低优先级中断，那么在高优先级中断执行完毕后直接去执行低优先级中断，低优先级中断执行完毕后才将高优先级中断压入堆栈的8个寄存器数据弹出，这中间就减少了高

优先级中断出栈以及低优先级入栈的过程。

Cortex内核的中断向量表可以通过编程重新映射到其它的地址空间，它的最开始保存的是栈指针，随后是各个中断向量。Cortex内核的中断向量里保存的是该中断对应的中断服务函数的地址，在发生中断时，硬件会自动跳转到相应的中断向量中的地址去执行中断服务程序，这一点与ARM7内核是不同的，ARM7内核的中断向量里保存的是跳转到该中断服务函数的指令，跳转到中断服务函数的动作是由软件完成的。

Cortex-M3有一个可选的存储器保护单元——MPU，芯片厂商可以根据自身设计规格来决定芯片上是否带有MPU单元，我选用的LM3S8962芯片上带有MPU单元。使用MPU可以指定一些区域不能被破坏，最常见的用法就是将不同任务的数据区使用MPU隔离开，避免一个任务出错影响到另一个任务。

Cortex内核采用了特权级和用户级，双堆栈指针MSP和PSP，以及MPU，这些机制增强了程序运行的稳定性。

最后，非常感谢“CM3权威指南”一书的译者，该书为我节约了非常多的查找cortex内核资料的时间，只是英文版的原书中使用了大量汇编语言的例子，这一点我不是很认同。我们开始接触一块芯片时没有必要去深入学习它的汇编语言，使用C语言就可以完成大多数的功能，如果我们不做系统设计，那么本节大多数的介绍都不需要关心，不需要关心是否使用了双堆栈，不需要关心外设是连在AHB总线上还是APB总线上，更不需要使用汇编语言去操作各个寄存器（只能使用汇编语言访问的寄存器除外），使用C语言就足够了。在一些群里发现一些朋友认为学习芯片就是学习汇编语言，我建议学习芯片是学习它的特性，使用它的特性，这些特性使用C语言就可以实现，汇编语言只占有很小的部分，不要将大把的精力浪费在芯片汇编语言的学习上。

如果本手册不是介绍操作系统的，需要深入了解一些汇编语言及芯片内核的特性，那么我是不会介绍本节的大部分内容的。也正是因为本手册的重点在于介绍操作系统的编写，因此我没有对cortex内核的特权+双堆栈+MPU特性加以利用。

第 2 节 开发环境

LM3S8962 和 STM32F103VB 芯片采用的都是 cortex 内核，因此在编程时非常相似，但这两个芯片又分属于不同的厂商，因此支持它们的库函数又有所差异，本节将介绍这两款芯片的开发环境，并对这两款芯片的开发板做一个简单介绍。

我们将继续在 Keil MDK4.20 的开发环境下开发程序，与前面章节所使用的 GNU 编译工具链所不同的是，本章开始我们将采用 Keil 自带的 RealVeiw 工具链进行编译，有关开发环境的配置请参考附录 4。

这两款芯片都提供了驱动库，将芯片外设的内部细节封装到了库函数内部，这样用户就可以直接使用库函数进行开发，加快开发速度。在 Wanlix 和 Mindows 的开发过程中就使用了芯片的库函数，其中所使用的 LM3S8962 芯片库版本是 7611，STM32F103VB 芯片库版本是 3.50。

LM3S8962 芯片采用编译好的 lib 库文件的方式开发操作系统，将操作系统源文件

（mindows 目录）、用户源文件（srccode 目录）和 lib 库文件（lib 目录）一起编译，生成最终的目标文件，而 STM32F103VB 芯片则直接使用库源文件，将操作系统源文件、用户源文件和库源文件一起编译，生成最终的目标文件。这两种方式不同，但结果是完全相同的，也是两种开发方式。

工程的目录结构没法发生变化，只是增加了一个 lib 目录用来存放芯片相关的库文件。

我使用的 LM3S8962 芯片的开发板是从 TI 申请到的，共有 2 块板子，一块主板功能较多，另有一块从板功能较少，这两块板子之间可以通过 can 总线进行通信，但在本手册中只使用了主板，只利用了主板上的液晶屏、按钮和 USB 接口。USB 接口复合了供电、仿真器及串口的功能，通过 USB 接口就可以实现对单板的供电，LM3S8962 开发板将仿真器集成到了开发板上，我们也可以使用 USB 接口直接仿真，单步跟踪程序的运行，非常方便，而且我们还可以通过 USB 虚拟串口的软件，将板上的 USB 接口虚拟成串口与计算机进行通信。

LM3S8962 主板的外观如下图所示：

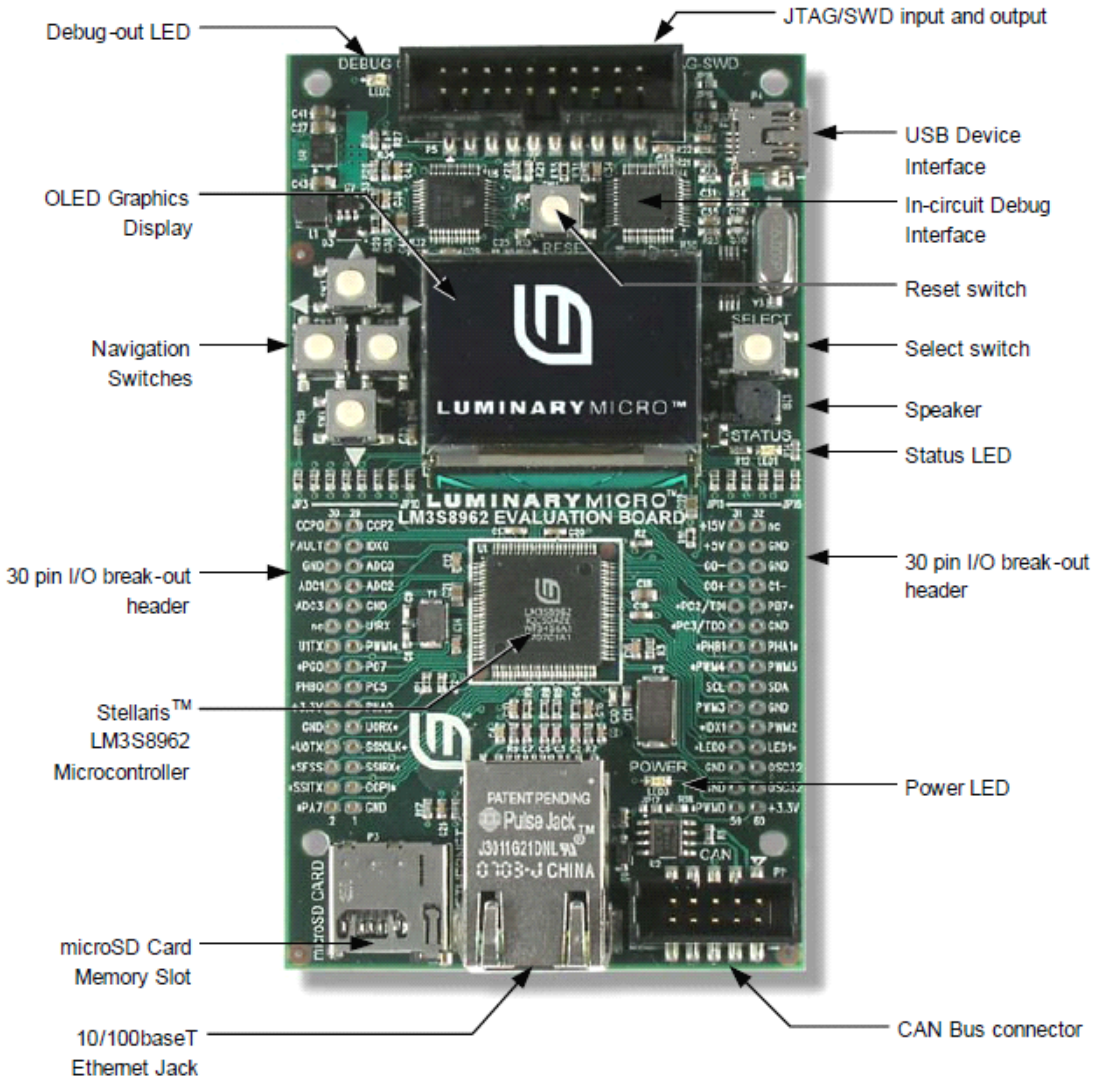


图 65　LM3S8962 开发板

我原本只是将 Wanlix 和 Mindows 移植到了 LM3S8962 开发板上，但发现更多的人使用的是 STM32F10X 的片子，为了使更多的人能自己感受到这 2 个操作系统的功能，当然也为了能推广这 2 个操作系统，我就在淘宝上又买了一块 STM32F103VB 的开发板，这块板子不

到 200 元，还比较便宜，如果你对我的这两个操作系统比较感兴趣，可以买一块和我一样的 STM32F103VB 开发板，或者使用其它的 STM32F10X 单板，在我发布的源代码基础上可以自己动手修改代码，试着增加一些功能，体验这两个操作系统。

在此声明一下，我与我使用的 STM32F103VB 开发板没有任何利益上的关系，只是看这个开发板比较便宜，而且也带有液晶屏、按钮和串口，在 5.5 节中，我将会以 Mindows 为基础编写一个俄罗斯方块的小游戏，需要使用到这些外设器件。STM32F103VB 这块板子做工有一点粗糙，有些器件是手焊的，但卖家服务态度还算不错，而且功能也没问题，如果大家也去买的话希望他们能便宜些卖给你们，这个板子在淘宝上的链接是 http://item.taobao.com/item.htm?id=10576675076，如果你还希望使用仿真器调试程序的话也可以购买一个支持 STM32F10X 芯片的仿真器，比如 ULINK、J-LINK 都可以。

STM32F103VB 开发板的外观如下图所示：

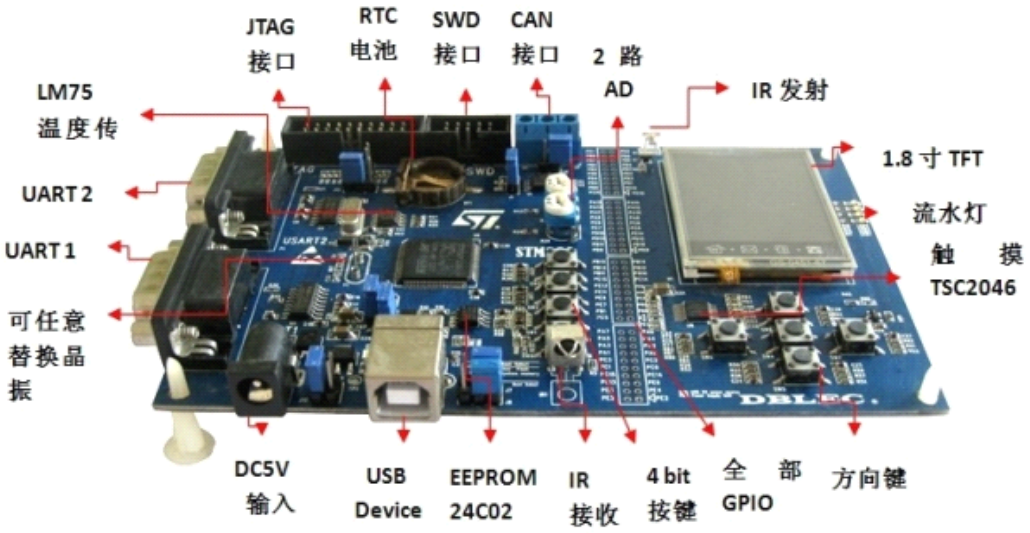


图 66　STM32F103VB 开发板

第 3 节 将 Wanlix 移植到 Cortex 芯片

前面基础知识介绍完毕，本节开始真刀真枪的移植代码了。本节将 Wanlix3.3 节的代码从 ARM7 内核移植到 TI 和 ST 的 cortex 内核的芯片上，移植完成后通过串口打印可以看到移植的效果。

C 语言具有很强的可移植性，因此在移植 Wanlix 的过程中 C 语言部分只有很少的修改，而且这些修改大部分是集中在外设驱动部分，因为不同芯片的外设不同，因此外设驱动部分的 C 语言修改是必然的。移植的主要修改点集中在汇编代码部分，因为汇编语言直接体现了芯片的特性，从 ARM7 内核到 cortex 内核结构发生了很大的变化，而且指令集也放生了变化，因此汇编代码部分需要全部重写，但整体框架和实现的方法并没有发生变化。

由于 LM3S8962 和 STM32F103VB 芯片同为 cortex 内核芯片，汇编部分都是一样的，它们之间代码的差异主要是体现在驱动库部分，因此只要一款芯片移植成功后，另一款芯片的移植也就非常容易了。

下面我们先来修改 LM3S8962 的代码，先从 wlx_core_a.asm 文件开始，这个文件是需要改动最大的文件。这个文件里只有两个函数，WLX_SwitchToTask 和 WLX_ContextSwitch，回想一下在第 3 章中的介绍，WLX_SwitchToTask 函数的功能是将 WLX_RootTask 任务在初始化时写入到堆栈的值恢复到 R0~R14 和 XPSR 中，然后跳转到保存在 R14 中的 WLX_RootTask 函数的指针，这样就进入操作系统状态了，开始执行第一个任务——WLX_RootTask 任务。WLX_ContextSwitch 函数的功能是备份、恢复系统在运行中需要切换的 2 个任务的上下文，实现任务切换。

在移植过程中，这 2 个函数的功能不会发生变化，我们只不过是需要使用 Thumb-2 指令集重新翻译一遍 ARM 指令集所做的工作。由于 Thumb-2 指令集与 ARM 指令集在细节上还有一些区别，因此在翻译时不可能与原来的指令一一对应，但变化不大，我们来看一下具体的实现过程：

```
00048  WLX_SwitchToTask
00049
00050      ;获取将要运行任务的指针
00051      LDR    R0, =guiNextTaskSp
00052      LDR    R13, [R0]
00053
00054      ;获取将要运行任务的堆栈信息并运行新任务
00055      POP    {R0}
00056      MSR    XPSR, R0
00057      POP    {R0 - R12}
00058      ADD    R13, #4
00059      POP    {PC}
```

首先我们可以看到汇编函数的形式与第三章中的不同了，这可不是因为指令集不同造成的，而是因为我们换了编译器，这两种编译器所规定的汇编书写格式不同。

00051 行，获取全局变量 guiNextTaskSp 的地址。

00052 行，获取全集变量 guiNextTaskSp 的内容，也就是将第一个需要运行的任务 WLX_RootTask 的 TCB 指针放入 SP 寄存器中，此时 SP 就指向了 WLX_RootTask 任务的堆栈，堆栈中存放的是初始化好的数据。

00055 行，此时 SP 指向了栈中 XPSR 的位置，可以参考图 13，使用 POP 指令从堆栈中弹出 XPSR 的数据存入 R0 寄存器，同时 SP 寄存器向栈顶移动 4 个字节，指向栈中 R0 的位置。POP 指令相当于是 LDMIA 指令。

00056 行，将栈中弹出的 XPSR 的数值通过 R0 写入到 XPSR 寄存器。

00057 行，从栈中弹出 R0~R12 寄存器的数值。

00058 行，运行到此处，SP 寄存器已经指向栈中 SP 的位置，此处将 SP+4，使其指向栈中 LR 的位置。

00059 行，使用 POP 指令将栈中 LR 的数据存入 PC 寄存器。此时已经完成了 XPSR、R0~R12 寄存器的恢复，在执行此行指令之前 SP 寄存器已经指向了栈中 LR 的位置，栈中 LR 中保存的是 WLX_RootTask 函数的指针，此行指令运行后 SP 寄存器会自动+4 正好指向栈顶，而 PC 寄存器中也装入了 WLX_RootTask 任务的函数指针，这样芯片内部所有的寄存器都被设置为 WLX_RootTask 任务初始化时的数值，也就开始运行 WLX_RootTask 任务了，从非操作系统状态进入操作系统状态。

这部分内容在第 3 章中会讲述的更详细些，如果你忘了，可以回去复习一下。

下面再来看看 WLX_ContextSwitch 函数：

```
00016  WLX_ContextSwitch
```

```
00017
00018       ;保存当前任务的堆栈信息
00019       SUB     R13, #0x8
00020       PUSH    {R0 - R12}
00021       MRS     R0, XPSR
00022       PUSH    {R0}
00023       MOV     R0, R13
00024       ADD     R0, #0x40
00025       STMDB   R0, {R0, R14}
00026
00027       ;保存当前任务的指针值
00028       LDR     R0, =gpuiCurTaskSpAddr
00029       LDR     R1, [R0]
00030       STR     R13, [R1]
00031
00032       ;获取将要运行任务的指针
00033       LDR     R0, =guiNextTaskSp
00034       LDR     R13, [R0]
00035
00036       ;获取将要运行任务的堆栈信息并运行新任务
00037       POP     {R0}
00038       MSR     XPSR, R0
00039       POP     {R0 - R12}
00040       ADD     R13, #4
00041       POP     {PC}
```

00019 行，执行此行指令之前 SP 指向切换前的任务的栈顶，从此位置向下将要保存该任务的 R14~R0 还有 XPSR 寄存器的数值，将 SP-8 使 SP 越过栈中 R14 的位置指向栈中 R13 的位置。

00020 行，将 R0~R12 寄存器的数值存入到栈中 R0~R12 寄存器的位置。ARM 芯片的栈是满栈，PUSH 指令相当于是 STMDB，执行 PUSH 指令会先指向下一个存储地址再压栈，。

00021 行，将 XPSR 寄存器的内容存入到 R0 寄存器。

00022 行，执行此行指令前 SP 指向栈中 R0 的位置，使用 PUSH 指令将 XPSR 寄存器的内容通过 R0 寄存器存入到栈中 XPSR 寄存器的位置。

00023 行，R0 寄存器的内容已经保存进堆栈了，此行将 SP 寄存器的内容存入 R0 寄存器。

00024 行，执行此行指令前 SP 也就是 R0 指向栈中 XPSR 的位置，R0+0x40 后 R0 指向了压栈前的栈顶。

00025 行，此时 R0 中保存的是 SP 的数值，此行指令将 LR 和 SP 压入栈中相应的位置，至此，切换前的任务的堆栈备份工作已经全部完成。23~25 行之所以使用 R0 来保存 SP，是因为 Thumb-2 指令集规定在对栈进行操作时不能使用 SP 寄存器，这点与 ARM 指令集是不同的。

00028~00030 行，将切换前的任务的栈指针存入到全局变量 gpuiCurTaskSpAddr 中，也就是存入到切换前的任务的 TCB 中的 uiTaskCurSp 变量中，至此，对切换前的任务的全部操作就完成了。

00033~00041 行与 WLX_SwitchToTask 函数的功能完全一样，是从切换后的任务的栈中恢复寄存器的数据，然后切换后的任务开始运行，完成任务切换。

至此，最主要的移植工作已经完成，我们再来看看余下的一些小改动点：

- 修改宏 MODE_USR 的数值，以便在创建任务时 XPSR 寄存器可以被正确初始化。
- Cortex 内核需要使用 8 字节对齐的栈，因此在 WLX_TaskTcbInit 函数里需要修改一下任务的初始栈顶，将其按 8 个字节对齐。
- 由于编译器对汇编语言规定不同，与汇编函数相关的一些声明需要修改。
- 由于芯片的指令速度不同，因此需要修改 DEV_DelayMs 函数里延迟的周期，以使该函数可

以延迟正确的时间。

上述的修改非常少，大家使用 Beyond Compare 工具对比一下就可以看得非常明白了。

除此之外，device.c 文件的改动还算是比较大，但这些改动全部是驱动程序的修改，与操作系统无关，只是使用 LM3S8962 的驱动函数替换原来 Aduc7024 的驱动函数，这里就不介绍了。

经过上面这些步骤，我们就将 Wanlix 从 ARM7 内核的 Aduc7024 芯片移植到了 cortex 内核的 LM3S8962 芯片上，而我们只需要在此基础上将 LM3S8962 芯片的驱动层函数更换为 STM32F103VB 芯片的驱动层函数就完成了将 Wanlix 移植到 STM32F103VB 芯片上的工作，当然由于这两款芯片的驱动库结构不同，会导致其它文件也会有一些小小的差异，但这些差异不属于操作系统的范围之内，这里就不一一介绍了。

编译修改后的代码，输出的截图如下：

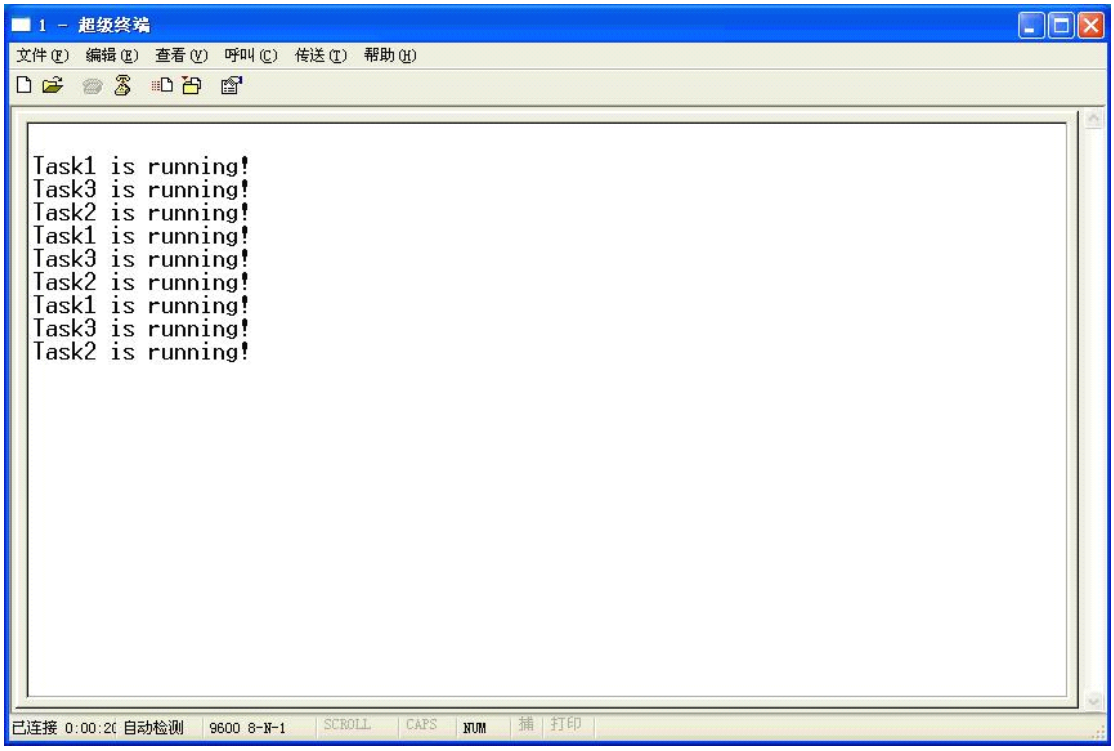


图 67 Wanlix 移植到 cortex 内核芯片上的打印信息

LM3S8962 和 STM32F103VB 的输出结果是一样的，因此在此只展示一个芯片的输出截图，大家可以到网站下载本节资源，里面包含这 2 个芯片的源代码和输出视频。

如果你的板子上的芯片与这两个芯片之一是同一系列的，如果板子有 UART1 接口，那么你也可以将本节的代码烧到你的板子上运行，亲身体验一下 Wanlix 的功能。

另外需要注意的是，我在移植代码过程中发现 STM32F103VB 芯片存在一个问题，可能是芯片的一个 bug，为了规避这个 bug 我在 STM3210E-EVAL.sct 文件里将 DEV_DelayMs 函数编译到了固定的 0x08019004 地址。我使用的 STM32F103VB 芯片有 128K 的 FLASH 程序空间，如果你使用的 ST 芯片小于 128K 的空间，那么你需要修改 STM3210E-EVAL.sct 文件中下面的代码才能编译过：

```
LR_IROM2 0x08019004 0x00000FFC  {    ; load region size_region
  ER_IROM2 0x08019004 0x00000FFC  {  ; load address = execution address
   unoptimize.o
  }
}
```

其中 0x08019004 是 unoptimize.c 文件被编译到的地址，即 LR_IROM2 段的起始地址，0x00000FFC 是 LR_IROM2 段的长度，你要保证 LR_IROM2 段不能超出你芯片的存储空间。

如果你还是没弄清楚应该怎样修改，那么最简单的方法就是将这段代码删掉，但这样有可能会触发这个 bug，当然你可能会看不出来 O(∩_∩)O。

至于这是个什么样的 bug，我会在 5.4 节发布后，将 Mindows 也移植到这 2 个 cortex 内核的芯片之后，专门整理一个文档对这个 bug 做一个系统的介绍，同时还会介绍一个破坏咬尾中断机制导致芯片跑飞的案例，这 2 个案例都是我在移植过程中遇到的，觉得还有一些借鉴意义，就写出来供大家参考。

第 4 节 将 Mindows 移植到 Cortex 芯片

上节我们成功的将 Wanlix 移植到了 LM3S8962 和 STM32F103VB 芯片上，本节我们将会将 Mindows4.10 节的程序也移植到这 2 个芯片上，与 Wanlix 不同的是，Mindows 采用了更复杂的调度机制，用到了 Tick 中断定时调度，而这两款芯片的中断机制使用了特有的硬件出入栈方式，这决定了 Mindows 的移植必然会与 Wanlix 有所不同。

Mindows 在 ARM7 内核中使用了 Tick 定时器硬件中断和 SVC 软中断来实现任务调度，需要保证调度任务的中断一定不能打断其它的中断，因为任务调度中断的功能是备份、恢复任务的上下文，如果任务调度中断发生在其它中断内，那么任务调度中断就会备份被打断的中断运行时的寄存器，并且恢复下一个最高优先级的任务，这样就从中断状态直接切换到了任务状态，也就是说被打断的中断没有执行完就异常退出了，这明显破坏了中断机制，会导致错误。ARM7 内核在硬件层次就不允许发生中断嵌套，一个中断执行时，其它中断就无法产生，因此这个问题 在 ARM7 内核中是不存在的。但 cortex 内核是支持中断嵌套的，因此我们必须将 cortex 内核调度中断的优先级设置为最低，来保证它不会打断其它的中断。但这样还存在一个问题——如果将 SVC 软中断设置为最低优先级，那么同一时刻发生的其它中断就会抢在 SVC 中断之前执行，而 SVC 中断如果不能立刻执行就会触发硬件异常。Cortex 内核专门提供了一个 PendSV 中断可以解决这个问题，PendSV 对比 SVC 的特点在于它可以 Pend，意思是说当 PendSV 中断发生时若有其它高优先级中断就绪那么它就等待，直到所有其它就绪的中断执行完，它才去执行，而且它既可以在中断中触发也可以在任务中触发，我们可以将 PendSV 的优先级设置为最低，在 Tick 中断内仅触发 PendSV 中断而不做任务调度，在需要使用 SVC 中断时也触发 PendSV 中断来代替直接产生 SVC 中断，而真正的任务调度只在 PendSV 中断内才进行，这样我们就可以将 Tick 和 SVC 调度中断统一到 PendSV 中断里了。

在 4.7 节讲二进制信号量时曾经讲过二进制信号量若在中断中激活了任务，那么这个任务必须要等到下个 Tick 中断到来时才能运行，原因就是中断中不能执行 SVC 立刻进行任务调度，而 PendSV 就可以做到，在中断中先触发 PendSV，当这个中断结束后立刻执行 PendSV 中断进行任务调度。

前面讲过 cortex 内核的中断会自动备份、恢复 XPSR、PC、LR、R12、R3、R2、R1 和 R0 这

8 个寄存器的数据，而 Mindows 则正是利用中断来进行任务上下文切换的，因此将 Mindows 从 ARM7 内核移植到 cortex 内核，任务调度中断部分是需要重新设计的。

进中断前，硬件会自动将这 8 个寄存器入栈，出中断时，硬件会自动从栈中恢复这 8 个寄存器的内容，这个过程对于软件来说是透明的，软件应该不需要做任何特殊的处理。但在中断调度函数里需要在任务 TCB 中的寄存器组与芯片寄存器之间备份、恢复寄存器的数据，中断调度函数执行前后的寄存器中的数值是不同的，是不同任务的，但中断出去时又会从当前栈中弹出 8 个数值存入上述的 8 个寄存器中，导致这 8 个寄存器的数据没有被正确恢复为下个运行任务的数值，因此采用原有 ARM7 内核的方式来备份、恢复寄存器组就会有问题了。

比如说从任务 A 切换到任务 B，ARM7 内核的调度机制会将当前寄存器的数值保存到任务 A 的 TCB 中，然后再从任务 B 的 TCB 中恢复寄存器的数值，这样就完成了任务切换。而 cortex 内核在进入中断前会将上述 8 个寄存器的数值保存任务 A 的栈中，然后进入中断，中断服务函数会将当前寄存器的数值保存到任务 A 的 TCB 中，然后再从任务 B 的 TCB 中恢复寄存器的数值，然后退出中断，此时 SP 指向了任务 B 的栈，但在退出中断时硬件会自动从当前栈中弹出 8 个寄存器的数值，而从任务 B 中弹出的都是无效数值，因此会导致软件跑飞。

为了解决这个问题，我们需要在寄存器备份、恢复之后多做一样工作，需要将上述 8 个寄存器的数值压入任务 B 的栈中，这样退出中断时，硬件就可以正确的取出这些寄存器的数值了。

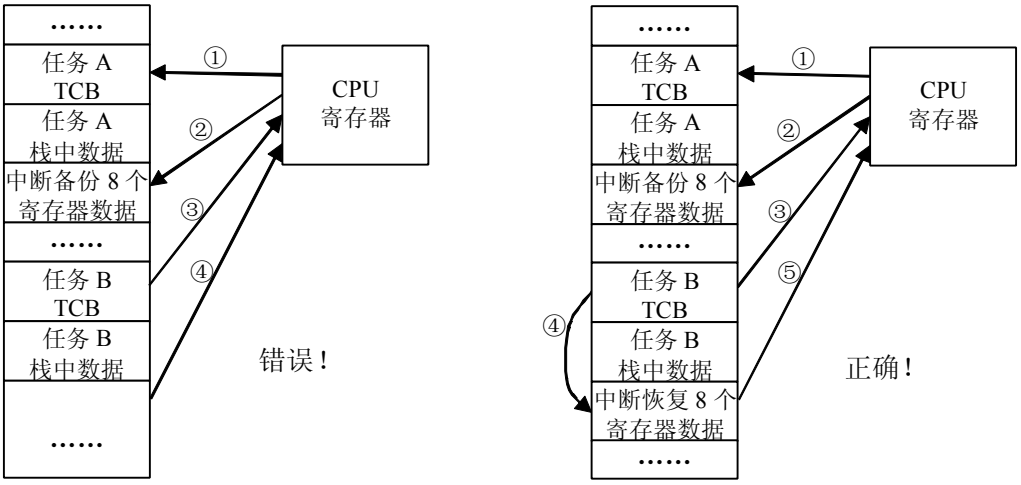


图 68　cortex 内核任务调度中断

在发生中断时，存入 LR 寄存器的不是返回地址，而是标示着返回时使用的模式和栈寄存器，因此这个数值也需要备份，我们需要在 TCB 的寄存器组结构中再增加一个 uiExc_Rtn 变量用来保存这个数值：

```
typedef struct stackreg
{
    U32 uiXpsr;
    U32 uiR0;
    U32 uiR1;
    U32 uiR2;
    U32 uiR3;
    U32 uiR4;
    U32 uiR5;
    U32 uiR6;
    U32 uiR7;
    U32 uiR8;
    U32 uiR9;
```

```
    U32 uiR10;
    U32 uiR11;
    U32 uiR12;
    U32 uiR13;
    U32 uiR14;
    U32 uiR15;
    U32 uiExc_Rtn;
}STACKREG;
```

在初始化堆栈的函数 MDS_TaskStackInit 里需要将 uiExc_Rtn 变量初始化为任务开始运行时使用的模式和栈寄存器，并且将上述 8 个寄存器压入栈中，以便第一次运行这个任务时可以正确取到数值，下面来看这个函数的代码：

```
00011  void MDS_TaskStackInit(M_TCB* pstrTcb, VFUNC vfFuncPointer)
00012  {
00013      STACKREG* pstrRegSp;
00014      U32* puiStack;
00015
00016      pstrRegSp = &pstrTcb->strStackReg;          /* 寄存器组地址 */
00017
00018      /* 对 TCB 中的寄存器组初始化 */
00019      pstrRegSp->uiXpsr = MODE_USR;               /* XPSR */
00020      pstrRegSp->uiR0 = 0;                        /* R0 */
00021      pstrRegSp->uiR1 = 0;                        /* R1 */
00022      pstrRegSp->uiR2 = 0;                        /* R2 */
00023      pstrRegSp->uiR3 = 0;                        /* R3 */
00024      pstrRegSp->uiR4 = 0;                        /* R4 */
00025      pstrRegSp->uiR5 = 0;                        /* R5 */
00026      pstrRegSp->uiR6 = 0;                        /* R6 */
00027      pstrRegSp->uiR7 = 0;                        /* R7 */
00028      pstrRegSp->uiR8 = 0;                        /* R8 */
00029      pstrRegSp->uiR9 = 0;                        /* R9 */
00030      pstrRegSp->uiR10 = 0;                       /* R10 */
00031      pstrRegSp->uiR11 = 0;                       /* R11 */
00032      pstrRegSp->uiR12 = 0;                       /* R12 */
00033      pstrRegSp->uiR13 = (U32)pstrTcb - 32;       /* R13 */
00034      pstrRegSp->uiR14 = (U32)MDS_TaskSelfDelete; /* R14 */
00035      pstrRegSp->uiR15 = (U32)vfFuncPointer;      /* R15 */
00036      pstrRegSp->uiExc_Rtn = RTN_THREAD_MSP;      /* EXC_RETURN */
00037
00038      /* 构造任务初始运行时的堆栈，该堆栈在任务运行时由硬件自动取出 */
00039      puiStack = (U32*)pstrTcb;
00040      *(--puiStack) = pstrRegSp->uiXpsr;
00041      *(--puiStack) = pstrRegSp->uiR15;
00042      *(--puiStack) = pstrRegSp->uiR14;
00043      *(--puiStack) = pstrRegSp->uiR12;
00044      *(--puiStack) = pstrRegSp->uiR3;
00045      *(--puiStack) = pstrRegSp->uiR2;
00046      *(--puiStack) = pstrRegSp->uiR1;
00047      *(--puiStack) = pstrRegSp->uiR0;
00048  }
```

00032 行之前没有变化，不再介绍。

00033 行，任务初始化时会预先存入 8 个寄存器，因此需要将栈指针向下移 32 个字节，存入到寄存器组的 SP 中。

00034~00035 行，没有变化，不再介绍。

00036 行，任务以 thread 模式运行，使用 MSP 堆栈。如果想使用其它的状态开始任务的运

行，那么需要修改此寄存器的初始值。

00039~00047 行，在栈中初始化 8 个寄存器。任务是在中断中开始第一次运行的，备份完上一个任务的寄存器，然后取出这个任务的寄存器数值，在中断返回时硬件会自动从当前栈中弹出 8 个寄存器的数据，因此需要在任务初始化时就预先装入这 8 个寄存器的数值。这个过程只对操作系统的第一个任务 root 任务例外，因为 root 任务是由 MDS_SwitchToTask 函数触发的，不是在中断中进入的，在介绍 MDS_SwitchToTask 函数时再详细说明。

下面来看看 MDS_SwitchToTask 函数的代码：

```
00097  MDS_SwitchToTask
00098
00099      ;获取将要运行任务的指针
00100      LDR    R0, =gpstrNextTaskSpAddr
00101      LDR    R13, [R0]
00102
00103      ;获取将要运行任务的堆栈信息并运行新任务
00104      ADD    R13, #0x40
00105      LDMIA  R13, {R0}
00106      SUB    R13, #0x40
00107      STMDB  R13, {R0}
00108      POP    {R0}
00109      MSR    XPSR, R0
00110      POP    {R0-R12}
00111      ADD    R13, #0x4
00112      POP    {LR}
00113      SUB    R13, #0x44
00114      POP    {PC}
```

00100~00101 行，将 SP 寄存器指向将要运行任务的寄存器组。

00104 行，执行此指令前 SP 指向新任务寄存器组中 XPSR 的位置，执行此指令后 SP 指针向上移 16 个寄存器位置，指向寄存器组中 R15 的位置。

00105 行，将寄存器组中 R15 的数值装入 R0 寄存器。

00106 行，将 SP 指向寄存器组中 XPSR 的位置。

00107 行，将 R0 内的数据装入寄存器组中 XPSR 的下一个栈内位置，也就是将初始化的 PC 数值装入栈中预先初始化的 XPSR 的位置。由于 MDS_SwitchToTask 函数是在非中断状态下启动 root 任务，因此硬件不会从预先存入的 8 个寄存器中恢复寄存器的数值，对这个函数来说 XPSR 的位置用来临时保存初始化时 PC 寄存器的数值，这 8 个预先存入的数值对这个函数来说是没有用的。

00108 行，此时 SP 指向寄存器组中 XPSR 的位置，此条指令将寄存器组中 XPSR 的数值装入 R0 寄存器。

00109 行，将初始化时的 XPSR 数值写入 XPSR 寄存器。

00110 行，将寄存器组中初始化的 R0~R12 数值恢复到 R0~R12 寄存器中。

00111 行，执行此指令前 SP 指向寄存器组中 SP 的位置，执行此指令后 SP 指向寄存器组中 LR 的位置。

00112 行，取出从寄存器组中取出 LR 寄存器的初始化数值。

00113 行，将 SP 指向预先存入的 8 个寄存器的 XPSR 的位置，该位置在 107 行存入了 PC 的数值。

00114 行，执行此指令前 XPSR、R0~R12 和 R14 寄存器已经恢复为了新任务的初始化数值，SP 指向的位置保存的是初始化的 PC 的数值，因此执行此指令后 SP 会向上移动一个寄存器位置，指向新任务的 TCB，同时 PC 寄存器也恢复为新任务初始化时的 PC 数值，至此，XPSR 以及

R0~R15 寄存器就全部恢复为了任务初始化时的数值，操作系统的第一个任务 root 任务就开始运行了。

root 任务所初始化的 SP 以及 8 个预先存入的寄存器数值并没有使用，root 任务初始栈的位置仍指向 TCB，任务初始化时预先存入的 8 个寄存器数值变为无效值。而其它任务是在中断中开始运行的，在任务切换函数 MDS_PendSvContextSwitch 执行之后，SP 指针指向预先存入的 8 个寄存器的 R0 的位置，中断返回时由硬件自动取出这 8 个寄存器的数值，SP 又指向了新任务的 TCB 位置。

MDS_PendSvContextSwitch 函数实现任务上下文切换的功能，它在 PendSV 中断中被调用，下面来看看它的代码：

```
00054  MDS_PendSvContextSwitch
00055
00056      ;保存接口寄存器
00057      PUSH   {R14}
00058
00059      ;调用 C 语言任务调度函数
00060      LDR    R0, =MDS_TaskSched
00061      ADR.W  R14, {PC} + 0x07
00062      BX     R0
00063
00064      ;保存当前任务的堆栈信息
00065      MOV    R14, R13
00066      LDR    R0, =gpstrCurTaskSpAddr
00067      LDR    R13, [R0]
00068      ADD    R14, #0x20
00069      LDMIA  R14, {R0}
00070      STMIA  R13!, {R0}
00071      SUB    R14, #0x1C
00072      LDMIA  R14, {R0 - R3, R12}
00073      STMIA  R13!, {R0 - R12, R14}
00074      ADD    R14, #0x14
00075      LDMIA  R14, {R0 - R1}
00076      STMIA  R13!, {R0 - R1}
00077      SUB    R14, #0x18
00078      LDMIA  R14, {R0}
00079      STMIA  R13, {R0}
00080
00081      ;任务调度完毕，恢复将要运行任务现场
00082      LDR    R0, =gpstrNextTaskSpAddr
00083      LDR    R1, [R0]
00084      ADD    R1, #0x14
00085      LDMIA  R1!, {R4 - R11}
00086      ADD    R1, #0x4
00087      LDMIA  R1, {R13}
00088      ADD    R1, #0xC
00089      LDMIA  R1, {R0}
00090      BX     R0
```

00054 行，将 LR 寄存器保存到堆栈中。LR 在被保存到寄存器组中之前需要使用它临时存放数据，因此需要将它先保存到栈中以防止被破坏。对比 ARM7 内核，我们并没有保存 R0~R3 以及 R12 这 5 个接口寄存器，因为这 5 个寄存器在发生中断时被硬件自动保存到了栈中，因此在此就不需要额外保存了。

00060~00062 行，调用 MDS_TaskSched 函数，做任务上下文切换前的准备工作。这几行代码并没有发生什么变化，只将原来的 ADD 指令更换成了 ADR.W 指令，因为 ADD 指令在 Thumb2

指令集里可能会编译为 2 字节或 4 字节的长度，而 61 行需要将调用 MDS_TaskSched 函数返回后的地址，也就是 65 行的地址存入到 LR 寄存器中，如果使用 ADD 指令则 65 行的地址会出现不确定的情况，而 ADR.W 指令固定为 4 字节长度，因此可以精确定位到 65 行的地址。需要注意的是实际上只需要+6，但由于是 Thumb2 指令集，最低 bit 需要为 1，因此变成+7 了。

00065 行，将 SP 寄存器复制到 LR 寄存器中。

00066~00067 行，将 SP 指向切换前任务的 TCB 位置，也就是寄存器组中 XPSR 的位置。

00068 行，由于在 54 行压入了 LR 寄存器，因此在执行此指令前 LR 指向中断压入的 8 个寄存器中 R0 位置下面的地址，执行此指令将 LR 上移 8 个寄存器位置，指向栈中 8 个寄存器中 XPSR 的位置。

00069 行，取出中断压入的 XPSR 寄存器的数值，保存到 R0 寄存器中。

00070 行，执行此指令前 SP 指向寄存器组中 XPSR 的位置，此指令将 R0 中保存的 XPSR 存入寄存器组中 XPSR 的位置。

00071 行，执行此指令前 LR 指向栈中 8 个寄存器中 XPSR 的位置，此指令将 LR 下移 7 个寄存器位置，指向 8 个寄存器中 R0 的位置。

00072 行，从栈中恢复 R0~R3 以及 R12 寄存器，自此，R0~R12 寄存器的数值就全部恢复为了中断发生时的数值，LR 寄存器的数值为中断发生时 SP 寄存器的数值。

00073 行，执行此指令前 SP 指向寄存器组中 R0 的位置，此指令将中断发生时 R0~R12 以及 SP（存储在 LR 中）的数值保存到寄存器组中 R0~R12 和 SP 相应的位置。

00074 行，将 LR 从 8 个寄存器中 R0 的位置上移 5 个寄存器位置指向 8 个寄存器中 LR 的位置。

00075 行，将 8 个寄存器中的 LR 和 PC 复制到 R0 和 R1 寄存器中。

00076 行，将 R0 和 R1 寄存器中的数值保存到寄存器组中 LR 和 PC 的位置，也就是完成了 LR 和 PC 寄存器的备份。

00077 行，将 LR 寄存器下移 6 个寄存器位置，指向栈中在 54 行压入的 LR 的位置。

00078 行，将 56 行入栈的 LR 数据存入 R0 寄存器中。

00079 行，执行此指令前 SP 指向寄存器组中 Exc_Rtn 的位置，此指令将中断发生时保存在 LR 中的数据存入寄存器组中的 Exc_Rtn。自此，完成了 XPSR、R0~R15 所有寄存器和返回时使用的 Exc_Rtn 的备份工作，备份切换前任务寄存器的工作完成。

00082~00083 行，将 R1 指向切换后任务的 TCB 位置，也就是寄存器组中 XPSR 的位置。

00084 行，将 R1 寄存器上移 5 个寄存器位置，指向寄存器组中 R4 的位置。

00085 行，从寄存器组中取出 R4~R11 的数值存入到 R4~R11 寄存器中。

00086 行，将 R1 指向寄存器组中 SP 的位置。

00087 行，从寄存器组中取出 SP 的数值存入到 SP 寄存器中。

00088 行，将 R1 指向寄存器组中 Exc_Rtn 的位置。

00089 行，将寄存器组中的 Exc_Rtn 恢复到 R0 寄存器中。

00090 行，跳转到 R0 寄存器所指向的地址，完成任务上下文切换。

上面恢复的过程只恢复了 R4~R11 和 SP 寄存器，剩下的 XPSR、R0~R3、R12、LR 和 PC 会在出中断时由硬件自动从栈中恢复为该任务在上次切出去时保存的数值。

上面所介绍的这 3 个函数就是将 Mindows 从 ARM7 内核移植到 cortex 内核的关键步骤，由于将原来的 Tick 中断和 SVC 中断任务调度的功能合并到了 PendSV 中断中执行，因此这部分函数的调用结构发生了一些小变化，但结构的本质并没有发生变化，这部分不再详细解释，还有其它的小改动与操作系统也没有太大关系，多半是与芯片驱动相关的，也不再做详细解释，请大家自行对比代码。

编译修改后的代码，输出的截图如下：

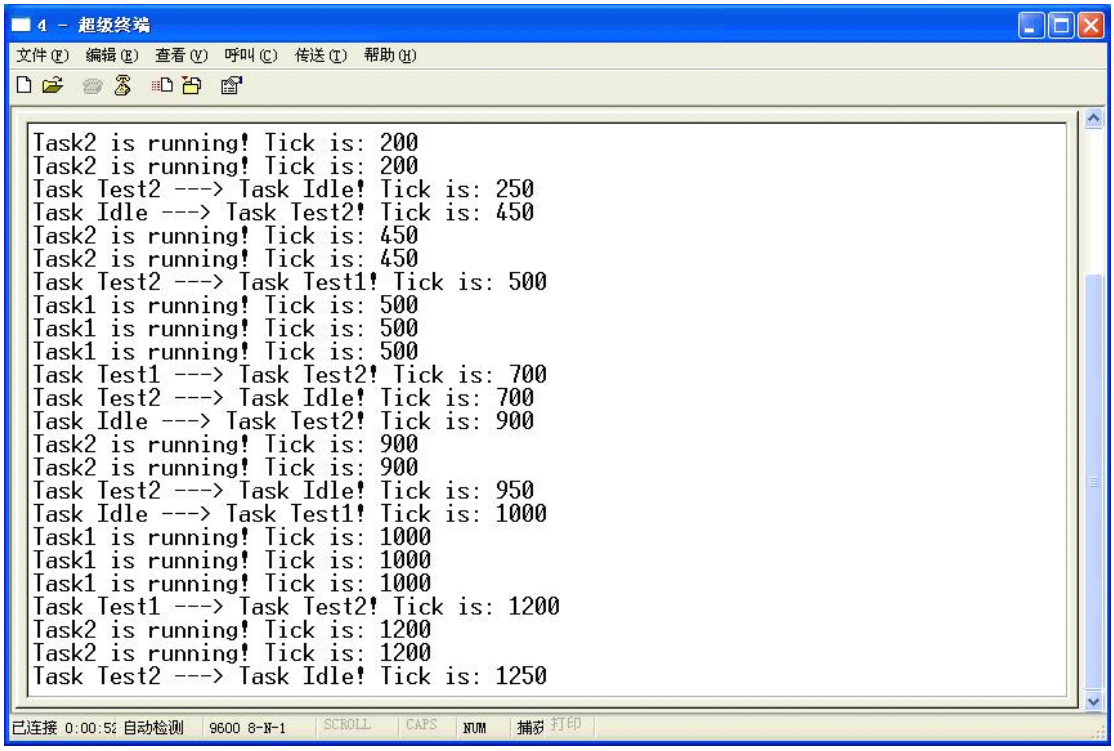


图 69 Mindows 移植到 cortex 内核芯片上的打印信息

视频等其它资源请到我的博客 blog.sina.com.cn/ifreecoding 下载，可以将本节的视频和串口文本与 4.10 的进行对比，可以看到两者是完全一样的，这也说明我们的移植是成功的。

第 5 节 在 Mindows 上编写俄罗斯方块的游戏

在第 4 章我们逐步开发了 Mindows 操作系统的一些功能，实现了实时抢占式调度，并拥有了信号量和队列等功能。在上一节，我们又将 Mindows 成功的移植到了具有 cortex 内核的 LM3S8962 和 STM32F103VB 芯片上，本节，我们将在这两款芯片的开发板上使用 Mindows 现有的功能开发一个俄罗斯方块的小游戏。

在 5.2 节我曾经介绍过我所使用的这 2 款芯片的开发板都有 LCD 液晶屏和方向按键，本节我们将使用这些资源做一个俄罗斯方块的小游戏。

俄罗斯方块的游戏我们应该都玩过，先将这个游戏的基本功能整理一下。我们最容易想到的是按键功能，需要能控制图形向下、向左和向右移动，还需要能旋转图形，当我们不按向下按键时图形还需要能自动向下走，当游戏结束时，我们需要一个按键可以重新开始游戏。在游戏过程中还需要遵守一些规则，比如说不同的图形互相之间不能重叠，图形不能移出屏幕的范围，当屏幕的一行被图形填满时本行需要被删除，并且上面所有的图像向下移动，还有最重要的一点，我们需要使用 LCD 将游戏的过程显示出来。

我们将这些需求做一个表格整理一下：

功能分类	子功能	描述
按键功能	开始键	按下开始键，游戏重新开始，所有状态归零
	旋转键	按下旋转键，图形可以旋转
	向下键	按下向下键，图形向下走
	向左键	按下向左键，图形向左走
	向右键	按下向右键，图形向右走
界面功能	全屏显示	需要能实时显示游戏画面
	下个图形显示	将屏幕分为左右两部分，左侧为游戏空间，右侧显示下个出现的图形
游戏功能	自动走	在没有按向下键时，图形应以一定速率自动向下走
	冲突检测	当图形移动、变形时不能与其它图形重叠
	下个图形开始	当图形向下走会发生冲突时对本图形的操作结束，下个图形从屏幕顶端进入屏幕，并更新屏幕右侧的下个图形
	删除一行	当图形向下走会发生冲突时对本图形的操作结束，检测是否有被图形占满的一行，若有需要删除此行
	删行闪烁	在删除一行时，需要反复改变被删除行的颜色，以呈现出闪烁效果，然后再删除此行
	图形下移	在一行被删除后，被删除行上面的所有图形都需要向下移一行
打印功能	任务切换打印	将任务切换过程从串口打印出来

表 12　俄罗斯方块游戏需求列表

表 12 中列出了我们需要做的功能，接下来我们就要想办法来实现这些功能。我们首先要解决的一个问题是我们必须要有一个办法能使用处理器来表示这些图形，这样才能对它们进行控制，并将它们显示到 LCD 上。LCD 中的每一个像素就是一个数据，代表着这个像素的颜色，因此我们可以在内存中用一个二维数组来对应 LCD 中的像素，数组的两个维度对应着 LCD 的 X 轴和 Y 轴，数组中元素的数值就对应着 LCD 中对应位置的像素的数值，数组元素为黑色像素的数值时代表 LCD 中对应的位置没有图形，其它数值代表有图形，这样就可以把对 LCD 中图形的操作转换为对数组数据的操作了，然后再以一定的频率将数组中的数据输出到 LCD 中，这样就实现用处理器控制 LCD 中的图形了。

经过上面的分析，我们就可以将图形映射为数组数据，我们只要专注于处理数组数据，至于 LCD 中的图形则只需要将数组数据刷到 LCD 上就可以了。由此可以想到的一个简单的处理方法是使用一个任务专门来处理数组数据，使用另一个任务专门来将数组数据刷到 LCD 上，至于按键功能则可以用另一个任务专门来检测按键的输入。这样的 3 个任务相互之间的耦合性很小，很适合使用并行的任务实现，按键任务负责检测按键输入，如有按键输入则向处理数据的任务传送按键值，处理数据的任务接收到按键值就对数组数据进行处理，而刷新 LCD 的任务只需要以固定的频率将数组数据刷新到 LCD 上就可以了。至于打印任务切换的功能还可以使用我们以前的方式实现，在任务切换时使用任务切换钩子函数将切换信息打印到内存中，再使用一个打印任务将内存中的数据打印到串口。

按键、处理、刷屏这 3 个用于实现游戏功能的任务结构关系如下图所示：

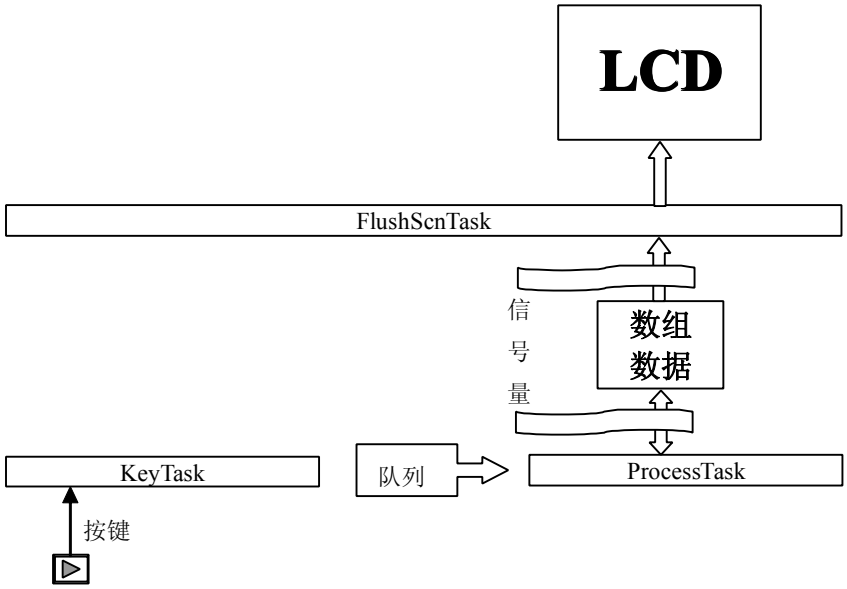


图 70　俄罗斯方块游戏任务结构图

KeyTask 任务周而复始的检测按键是否按下，由于对按键按下的频率有要求，因此这个任务需要以一定的间隔来读取按键值，在间隔期间内不读取按键值，这样也可以防止按键抖动的情况发生。当该任务读取到一个有效的按键值时，就将这个按键值压入队列发送给 ProcessTask 任务。

ProcessTask 任务周而复始的从队列里取消息，当队列为空时，该任务就会被队列阻塞，处于 pend 状态，直到队列里有了消息才被激活，然后从队列里取出按键值，根据按键值来对图形数组的数据做相应的操作。

FlushScn 任务则以一定的频率将图形数组中的数据刷新到 LCD 屏幕上。每个图形是由多个像素组成的，对图形处理、刷新的过程就是对图形中像素的处理、刷新过程，需要对多个像素进行多次操作才能完成对一个图形的操作。FlushScn 任务和 ProcessTask 任务是并行执行的，为避免这两个任务在对同一个图形的处理过程中相互干扰，就需要在这 2 个任务中使用信号量锁住对同一个图形的处理过程，使这两个任务对同一个图形的操作保持串行性，保证图形处理、显示时的完整性。

这 3 个任务的结构大致如下所示：

```
void TEST_KeyTask(void)
{
    while(1)
    {
        /* 任务以一定的间隔读取按键值 */
        MDS_TaskDelay();

        /* 读取按键值 */
        DEV_ReadKey();

        /* 将按键值放入队列 */
        MDS_QuePut();
    }
}

void TEST_ProcessTask(void)
{
    while(1)
    {
```

```
        /* 获取按键值 */
        MDS_QueGet();

        /* 获取信号量 */
        MDS_SemTake();

        /* 根据按键值对数组数据做相应处理 */
        TEST_KeyProcess();

        /* 释放信号量 */
        MDS_SemGive();
    }
}

void TEST_FlushScnTask(void)
{
    while(1)
    {
        /* 获取信号量 */
        MDS_SemTake();

        /* 刷新 LCD 屏幕 */
        TEST_FlushScn();

        /* 释放信号量 */
        MDS_SemGive();

        /* 任务以一定的间隔刷新 LCD */
        MDS_TaskDelay();
    }
}
```

KeyTask 任务和 FlushScn 任务比较简单，都是以固定的频率执行重复的动作，ProcessTask 任务比较复杂，是实现俄罗斯方块游戏的关键，下面我们再重点介绍一下这个任务。

ProcessTask 任务的主要功能是根据按键值来对数组数据做相应的处理，这其中涉及到图形移动、变形、冲突检测、删除等操作，实现了这些操作也就基本实现了俄罗斯方块这个游戏，我们先以 LM3S8962 为例讲解这些过程。

LM3S8962 板子上 LCD 的大小为 96×128 像素，每个像素只有 16 种灰度，因此只需要 4bits 就可以表示一个像素。这其中我们使用左边的 64×128 像素作为游戏区域，右边 28×128 像素作为下个图形显示区域，中间用 4×128 像素的线条作为左右部分的分隔线。左边游戏区域内的图形会不断的发生变化，因此我们需要使用一个 64×(128÷2)大小的字符型二维数组来表示游戏区域。中间分隔线不会变化，只需要在初始化时画出这条线就可以了，不需要在内存中保留相应的数组。右侧的下个图形显示区只是更新下个图像这一小块区域就可以了。俄罗斯方块游戏里共有 8 种基本的图形，部分图形经过旋转会表现出不同的形状，如下图所示：

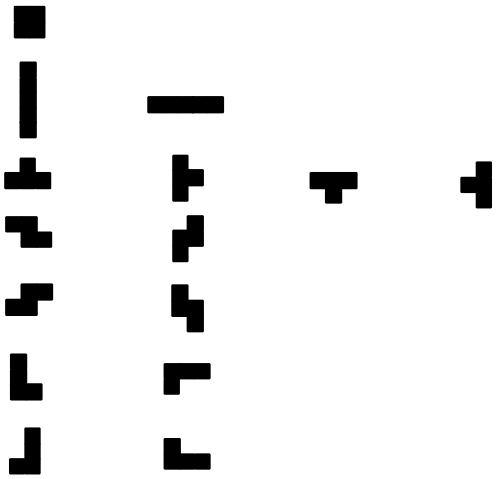


图 71　俄罗斯方块游戏图形

这些图形中每个基本的小方块占 4×4 个像素，最长的图形长度为 5×4=20 个像素，考虑到变形，我们需要使用 20×20 个像素的一块空间才能存放下任意一个图形，因此，我们只需要在内存中开辟一个 20×(20÷2)大小的字符型二维数组就可以表示任意一个图形了，更新屏幕右侧的下一个图形窗口只需要这么大小的一块内存就可以了。

当图形在游戏区域内向下移动的时候，我们还需要开辟一个 20×(20÷2)大小的字符型二维数组来表示当前的图形，当图形向下走一行时，先从游戏区域数组里将该图形清除掉，然后将当前图形数组在游戏区域数组内下移一行，如果下移后的当前图形数组与游戏区域数组没有图形数据冲突的话，就将下移后的当前图形数组写入游戏区域数组内，完成本次下移操作。如果有冲突的话，则说明当前图形已经不能再下移动了，将当前图形数组原地恢复到游戏区域数组内，再去判断当前图形所在的所有行内是否有填充满的整行数据，如果有的话就删除这些行，在删除的时候每隔一段时间变换一下这些行的颜色，以表现出删行时闪烁的效果。当删除一行后，需要将该行上面的所有图形数据都向下移动一行，并将下个图像数组复制到当前图形数组内，将当前数组的位置恢复到图形出现的最上方，再随机选择一个图形，将下个图形数组更新为该图形，完成本次操作。

其它的向左向右和变形操作也需要先从游戏区域数组内将当前图形清除掉，然后判断相应操作后是否会有图形冲突，如果没有冲突的话就将当前图形数组数据复制到操作后所在的游戏区域数组内的位置，完成本次操作，如果有冲突的话就将当前图形数组原地恢复到游戏区域数组内，结束本次操作。

上面就是有关俄罗斯方块游戏的基本介绍，代码就不具体介绍了，代码里都有注释，通过本节的介绍并结合代码应该就可以理解这个游戏的实现方法了。如果你感兴趣的话，可以再增加一个按键声音的功能，无非就是再创建一个播放声音的任务，当按键任务检测到按键时向声音任务发送一个消息激活声音任务，由声音任务播放按键声音，原来的软件结构基本不用改动。

下图是在 LM3S8962 单板上运行俄罗斯方块的图片，如果你想观看游戏视频的话请到我的博客 blog.sina.com.cn/ifreecoding，在 5.5 节里有视频可以观看。

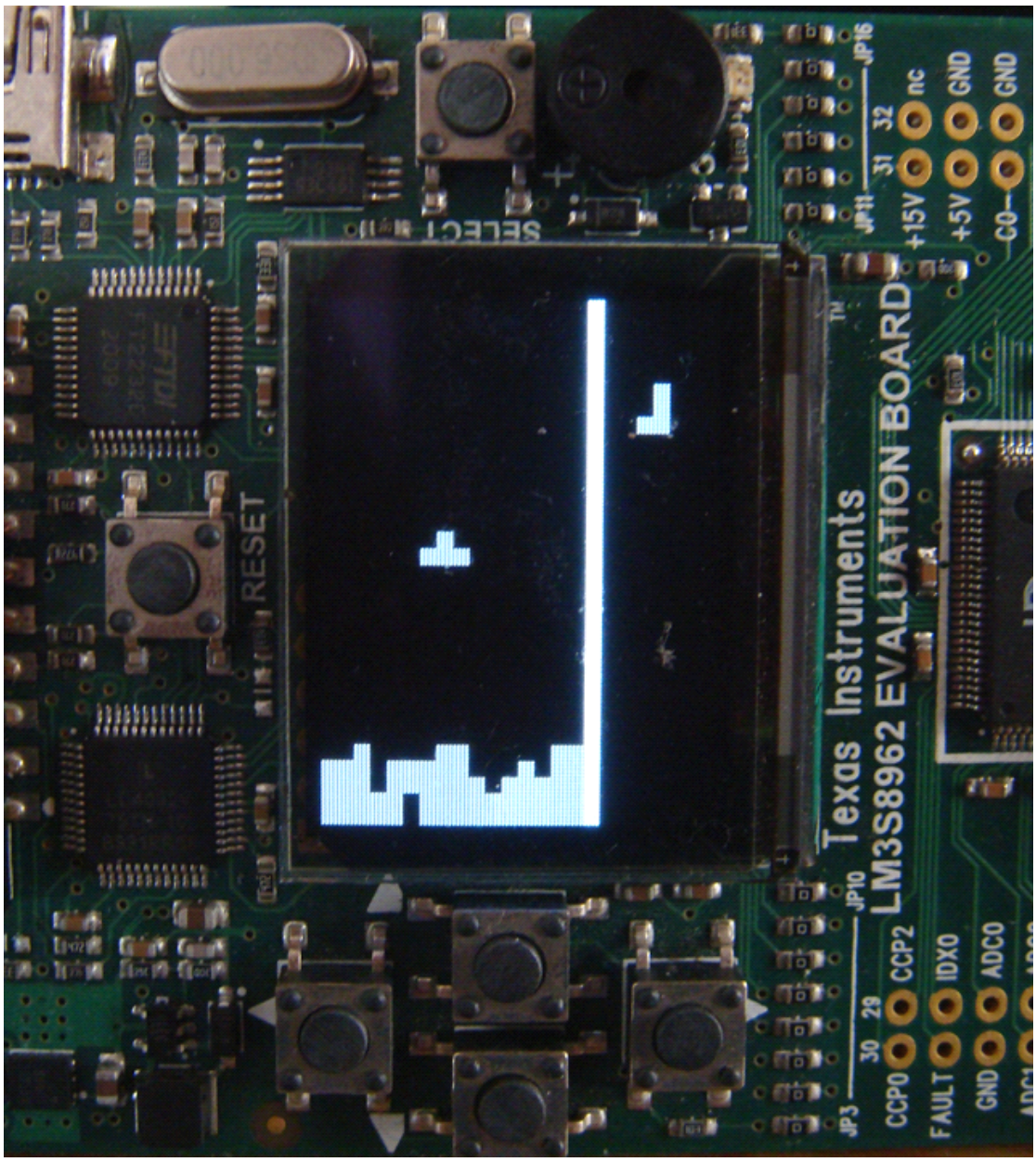


图 72　LM3S8962 单板运行俄罗斯方块游戏

LM3S8962 单板使用的 LCD 是 128×96 的分辨率，每个像素用 4bits 表示颜色，而 STM32F103VB 板子上使用的 LCD 是 160×128 的分辨率，每个像素需要用 16bits 表示颜色，这就是说 STM32F103VB 板子会比 LM3S8962 板子耗费更多的内存，而 STM32F103VB 芯片的内存又比较少，将俄罗斯方块游戏从 LM3S8962 单板移植到 STM32F103VB 单板上时内存就显得有些不够用了。在我们上述的设计中每个图形都是由 4×4 像素的基本小方块组成的，每次操作都是针对这个基本单元进行的，因此我们可以将这 16 个像素看做是一个像素，这样 128×96 个像素就可以缩减为 32×24 个像素了，所使用的内存也就大大减少了。

前面我们已经将 Mindows 操作系统移植到 STM32F103VB 板子上了，因此在这里仅需要修改 LCD 驱动程序以及与 LCD 操作相关的程序就可以将俄罗斯方块游戏从 LM3S8962 单板移植到 STM32F103VB 单板上，至于程序的整体结构则完全不用修改。另外，STM32F103VB 单板的 LCD 可以显示多种色彩，我们为每种图形分配一种颜色，这样看起来会更漂亮一些，代码很简单，不详细介绍了，请自行阅读代码。

下图是在 STM32F103VB 单板上运行俄罗斯方块的图片，在我的博客 5.5 节里有视频可以观看游戏过程。

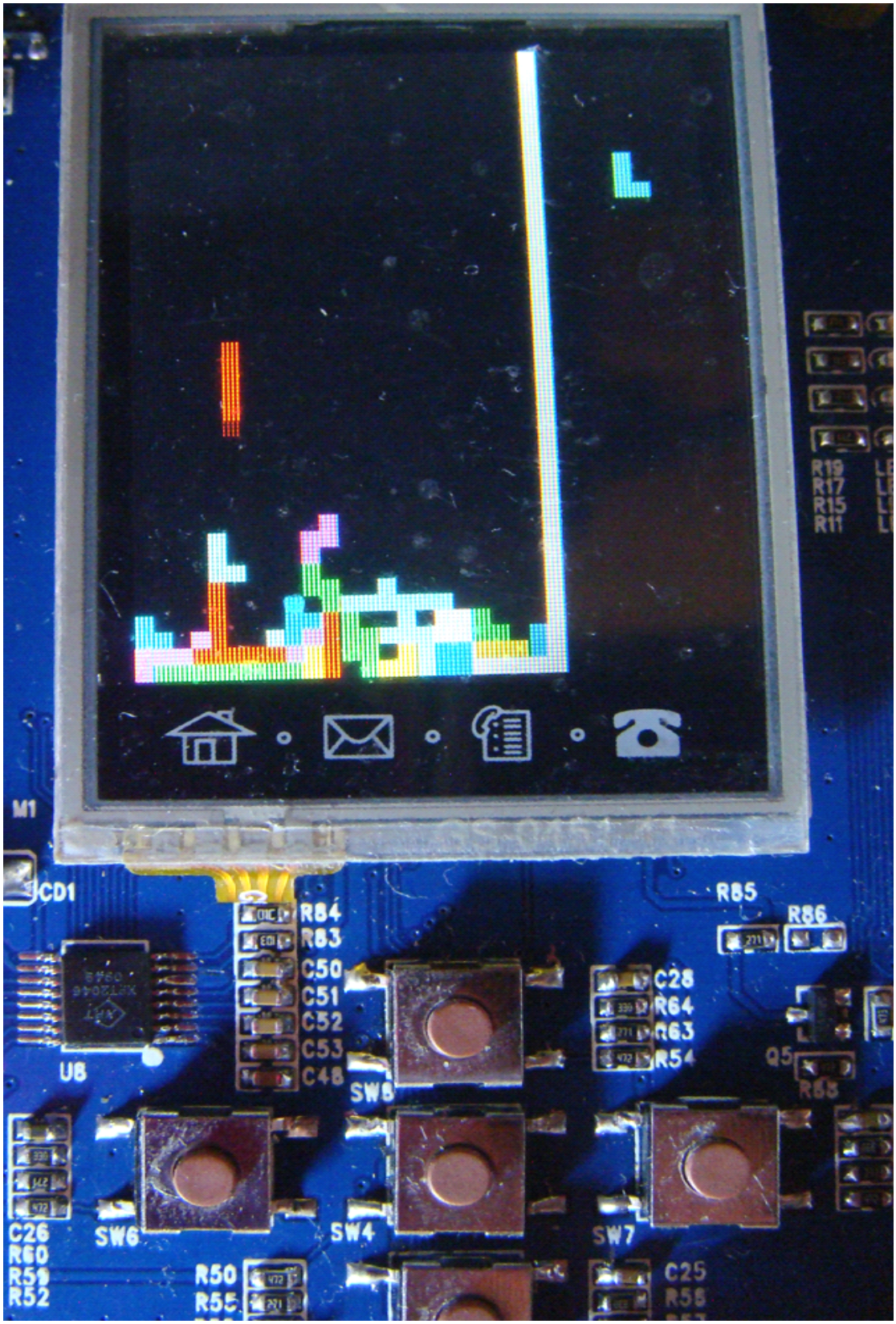


图 73　STM32F103VB 单板运行俄罗斯方块游戏

第 6 节 从堆申请内存

经过前面几节的开发我们已经将 wanlix 和 mindows 成功移植到了 cortex 内核的芯片上，并在 2 款 cortex 内核芯片上开发了一个俄罗斯方块游戏验证了 mindows 的任务调度、信号量和队列等功能。到目前为止我们在创建任务、信号量和队列时还需要为其静态申请内存，这样操作起来会显得有些麻烦，本节将增加自动从堆申请内存的功能。

这个功能实现起来很简单，即在使用 MDS_TaskCreate 函数创建任务、使用 MDS_SemCreate 函数创建信号量、使用 MDS_QueCreate 函数创建队列时，如果输入的内存参数为 NULL 则由创建函数在自己内部从堆申请所需要的内存，并任务、信号量或者队列时再释放这些内存就可以了。

C 语言库提供的 malloc、free 函数可以从堆里申请、释放内存，为使用这 2 个函数，需要包含 stdlib.h 头文件，本章代码编译时采用的是 Keil 自带的 RealVeiw 工具链，该工具链需要在启动文件 start.s 里按如下方式分配堆栈空间，代码如下：

```
00229   LDR     R0, =HeapMem
00230   LDR     R1, =(StackMem + Stack)
00231   LDR     R2, =(HeapMem + Heap)
00232   LDR     R3, =StackMem
```

00229 行，将堆的基址放入 R0 寄存器中，00230 行，将栈基址放入 R1 寄存器中，00231 行将堆的结束地址放入 R2 寄存器中，00232 行将栈的结束地址放入 R3 寄存器中，其中的 HeapMem、StackMem、Stack 和 Heap 定义如下：

```
Stack     EQU     0x00000200
Heap      EQU     0x00002000
StackMem  SPACE   Stack
HeapMem   SPACE   Heap
```

在这里定义了 Stack（栈）大小为 0x200 字节，这个栈是软件还未进入操作系统之前所使用的栈，进入操作系统后每个任务使用任务创建时所分配的栈。定义了 Heap（堆）大小为 0x2000 字节，malloc 所申请的内存就是从这个堆里申请的，也就是说任务使用 malloc 申请到的任务栈是从堆里获得的。StackMem 代表一块内存空间的起始地址，这块内存的大小为 Stack，也就是 0x200 字节。HeapMem 代表一块内存空间的起始地址，这块内存的大小为 Heap，也就是 0x2000 字节。

ARM 是递减栈，因此将栈的高地址分配给 R1 寄存器，而将栈的低地址分配给 R3 寄存器。堆是递增的，因此将堆的低地址分配给 R0 寄存器，将堆的高地址分配给 R2 寄存器。在这 4 个寄存器配置完成后就需要初始化 C 语言运行环境了，C 语言将通过这 4 个寄存器传进来的值来配置堆栈的范围。

下图 74 是堆、栈在内存中的分部情况，你也可以查看图 15，对比没有使用堆和和使用堆这 2 种情况下任务栈的分部情况。

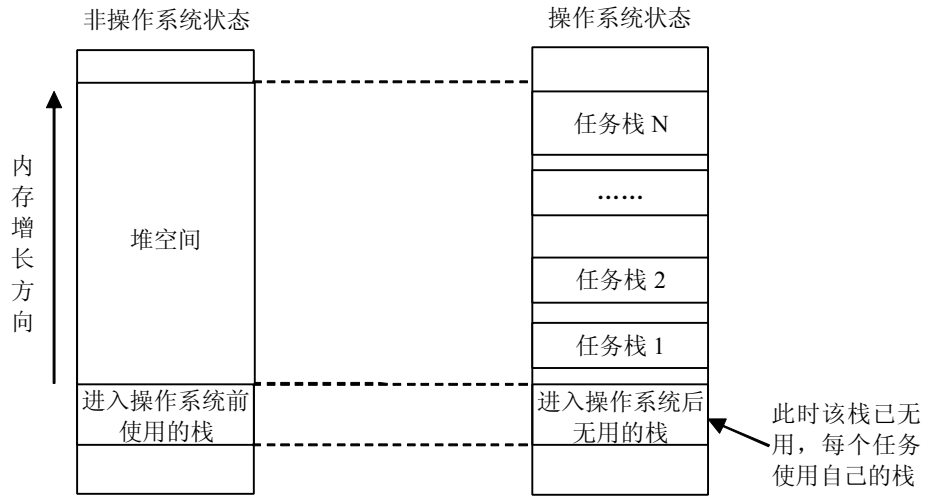


图 74　从堆中分配任务栈空间

由于是在创建任务、信号量和队列时申请的内存，因此在删除任务、信号量和队列时需要释放这些内存，因此需要在这 3 种结构中分别增加一个指针来记录申请到的内存地址，如果在创建时使用的是静态内存，不需要释放，就将这指针置为 NULL，如果是在创建时动态申请的内存，就将这指针置为申请到的内存地址，释放时就可以通过这个指针释放所申请的内存了。

```
/* TCB 结构体 */
typedef struct m_tcb
{
    STACKREG strStackReg;           /* 备份寄存器组 */
    M_TCBQUE strTcbQue;             /* TCB 结构队列 */
    M_TCBQUE strDelayQue;           /* delay 表队列 */
    M_SEM* pstrSem;                 /* 与任务相关的信号量指针 */
    U8* pucTaskName;                /* 任务名称指针 */
    U8* pucTaskStack;               /* 创建任务时的堆栈地址 */
    U32 uiTaskFlag;                 /* 任务标志 */
    U8 ucTaskPrio;                  /* 任务优先级 */
    M_TASKOPT strTaskOpt;           /* 任务参数 */
    U32 uiStillTick;                 /* 延迟到的时间 */
}M_TCB;

/* SEM 结构体 */
typedef struct m_sem
{
    M_TASKSCHEDTAB strSemTab;       /* 信号量调度表 */
    U32 uiCounter;                  /* 信号量计数值 */
    U32 uiSemOpt;                   /* 信号量参数 */
    U8* pucSemMem;                  /* 创建信号量时的内存地址 */
    struct m_tcb* pstrSemTask;      /* 获取到互斥信号量的任务 */
}M_SEM;

typedef struct m_que    /* 队列结构 */
{
    M_CHAIN strChain;   /* 队列链表 */
    M_SEM strSem;       /* 队列信号量 */
    U8* pucQueMem;      /* 创建队列时的内存地址 */
}M_QUE;
```

这其中任务与信号量以及队列在处理上有一点小区别，任务是使用 TCB 中的 uiTaskFlag 来标记是否需要释放内存，而不是用指针是否为 NULL 来区分。

我们先来看看创建任务函数 MDS_TaskCreate 的实现方式，MDS_TaskCreate 函数的原形如下：

```
M_TCB* MDS_TaskCreate(U8* pucTaskName, VFUNC vfFuncPointer, U8* pucTaskStack,
                      U32 uiStackSize, U8 ucTaskPrio, M_TASKOPT* pstrTaskOpt)
```

其中 pucTaskStack 是被创建的任务所使用的任务栈指针，如果该指针不为 NULL，则说明任务栈在创建任务前就已经准备好了，创建任务时可以直接使用，如果为 NULL，则说明在创建任务时需要由 MDS_TaskCreate 函数自动申请内存作为任务栈。MDS_TaskCreate 函数将 pucTaskStack 参数传递给了 MDS_TaskTcbInit 函数，由 MDS_TaskTcbInit 函数判断是否需要申请内存，如下所示：

```
00220  M_TCB* MDS_TaskTcbInit(U8* pucTaskName, VFUNC vfFuncPointer, U8* pucTaskStack,
00221                         U32 uiStackSize, U8 ucTaskPrio, M_TASKOPT* pstrTaskOpt)
00222  {
.....      .....
00229
00230      /* 锁中断，防止其它任务影响 */
00231      (void)MDS_IntLock();
00232
00233      /* 若传入堆栈为空，则由任务自己申请内存 */
00234      if(NULL == pucTaskStack)
00235      {
00236          pucTaskStack = malloc(uiStackSize);
00237          if(NULL == pucTaskStack)
00238          {
00239              /* 返回前解锁中断 */
00240              (void)MDS_IntUnlock();
00241
00242              /* 申请不到内存，返回空指针 */
00243              return (M_TCB*)NULL;
00244          }
00245
00246          /* 将任务标志置为堆栈申请状态 */
00247          uiTaskFlag = TASKSTACKFLAG;
00248      }
00249      else /* 是传入的堆栈，先将任务标志置为空 */
00250      {
00251          uiTaskFlag = 0;
00252      }
00253
.....      .....
00259
00260      /* 保存任务堆栈起始地址 */
00261      pstrTcb->pucTaskStack = pucTaskStack;
00262
.....      .....
00268
00269      /* 将任务标志置为申请堆栈的状态 */
00270      pstrTcb->uiTaskFlag |= uiTaskFlag;
00271
.....      .....
00322
00323      (void)MDS_IntUnlock();
```

```
00324
......      ......
00326  }
```

这段代码很简单，若 pucTaskStack 为 NULL，则使用 malloc 从堆中申请任务栈，将任务栈指针存入 TCB 中的 pucTaskStack 中，并将 TCB 中的 uiTaskFlag 标志置为申请堆栈状态。细节就不介绍了，其它部分代码相对原来的代码没什么变化。需要注意是，malloc 与 free 函数是不可重入函数，因此扩大了锁中断的范围，将它们也包含进去了。

删除任务函数 MDS_TaskDelete 更为简单，如果 TCB 中的 uiTaskFlag 标志被置为申请堆栈状态，那么就释放掉存放在 TCB 中 pucTaskStack 中的指针，代码如下，仅列出修改部分：

```
00121  U32 MDS_TaskDelete(M_TCB* pstrTcb)
00122  {
......      ......
00174
00175      /* 如果是任务自己申请的堆栈，则需要释放 */
00176      if(TASKSTACKFLAG == (pstrTcb->uiTaskFlag & TASKSTACKFLAG))
00177      {
00178          free(pstrTcb->pucTaskStack);
00179      }
00180
......      ......
00197  }
```

有关信号量、队列创建及删除函数改动的代码如下，不再详细介绍，很容易理解：

```
00024  M_SEM* MDS_SemCreate(M_SEM* pstrSem, U32 uiSemOpt, U32 uiInitVal)
00025  {
......      ......
00054
00055      /* 传入指针为空，需要自己申请内存 */
00056      if(NULL == pstrSem)
00057      {
00058          (void)MDS_IntLock();
00059
00060          /* 多申请 3 个字节，保证 pstrSem 开始的地址对齐到 4 字节 */
00061          pucSemMemAddr = malloc(sizeof(M_SEM) + 3);
00062          if(NULL == pucSemMemAddr)
00063          {
00064              (void)MDS_IntUnlock();
00065
00066              /* 申请不到内存，返回失败 */
00067              return (M_SEM*)NULL;
00068          }
00069
00070          (void)MDS_IntUnlock();
00071
00072          /* 将信号量结构对齐到 4 字节 */
00073          pstrSem = (M_SEM*)(((U32)pucSemMemAddr + 3) & 0xFFFFFFFC);
00074      }
00075      else /* 是传入的信号量内存指针 */
00076      {
00077          /* 将信号量内存地址置为空 */
00078          pucSemMemAddr = (U8*)NULL;
00079      }
```

```
00080
.....       ......
00089
00090       /* 保存信号量内存的起始地址 */
00091       pstrSem->pucSemMem = pucSemMemAddr;
00092
.....       ......
00097   }

00619   U32 MDS_SemDelete(M_SEM* pstrSem)
00620   {
.....       ......
00636
00637       /* 如果是创建信号量函数申请的信号量内存，则需要释放 */
00638       if(NULL != pstrSem->pucSemMem)
00639       {
00640           (void)MDS_IntLock();
00641
00642           free(pstrSem->pucSemMem);
00643
00644           (void)MDS_IntUnlock();
00645       }
00646
.....       ......
00648   }

00017   M_QUE* MDS_QueCreate(M_QUE* pstrQue, U32 uiSemOpt)
00018   {
.....       ......
00027
00028       /* 传入指针为空，需要自己申请内存 */
00029       if(NULL == pstrQue)
00030       {
00031           (void)MDS_IntLock();
00032
00033           /* 多申请3个字节，保证pstrQue开始的地址对齐到4字节 */
00034           pucQueMemAddr = malloc(sizeof(M_QUE) + 3);
00035           if(NULL == pucQueMemAddr)
00036           {
00037               (void)MDS_IntUnlock();
00038
00039               /* 申请不到内存，返回失败 */
00040               return (M_QUE*)NULL;
00041           }
00042
00043           (void)MDS_IntUnlock();
00044
00045           /* 将队列结构对齐到4字节 */
00046           pstrQue = (M_QUE*)(((U32)pucQueMemAddr + 3) & 0xFFFFFFFC);
00047       }
00048       else /* 是传入的队列内存指针 */
00049       {
00050           /* 将队列内存地址置为空 */
00051           pucQueMemAddr = (U8*)NULL;
00052       }
00053
.....       ......
00055
00056       /* 保存队列内存的起始地址 */
00057       pstrQue->pucQueMem = pucQueMemAddr;
```

```
00058
……      ……
00074  }

00154  U32 MDS_QueDelete(M_QUE* pstrQue)
00155  {
……      ……
00167
00168      /* 如果是创建队列函数申请的队列内存，则需要释放 */
00169      if(NULL != pstrQue->pucQueMem)
00170      {
00171          (void)MDS_IntLock();
00172
00173          free(pstrQue->pucQueMem);
00174
00175          (void)MDS_IntUnlock();
00176      }
……      ……
00179  }
```

本节的任务、队列都由创建函数自己申请内存，我们仍使用 3 个测试任务 TEST_TestTask1、TEST_TestTask2 和 TEST_TestTask3，其中 TEST_TestTask1 是由 MDS_RootTask 任务创建的，它是一个死循环任务，周而复始的创建 TEST_TestTask2 和 TEST_TestTask3 任务，TEST_TestTask2 和 TEST_TestTask3 任务运行一次就自结束，这样 TEST_TestTask2 和 TEST_TestTask3 任务就会不停的申请释放任务栈。

```
00013  void TEST_TestTask1(void)
00014  {
00015      while(1)
00016      {
00017          DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00018                          MDS_SystemTickGet());
00019
00020
00021          /* 创建任务 2 */
00022          (void)MDS_TaskCreate((U8*)"Test2", (VFUNC)TEST_TestTask2, (U8*)NULL,
00023                               TASKSTACK, 1, (M_TASKOPT*)NULL);
00024
00025          /* 创建任务 3 */
00026          (void)MDS_TaskCreate((U8*)"Test3", (VFUNC)TEST_TestTask3, (U8*)NULL,
00027                               TASKSTACK, 3, (M_TASKOPT*)NULL);
00028
00029          DEV_DelayMs(2000);
00030
00031          (void)MDS_TaskDelay(300);
00032      }
00033  }

00040  void TEST_TestTask2(void)
00041  {
00042      DEV_PutStrToMem((U8*)"\r\nTask2 is running! Tick is: %d",
00043                      MDS_SystemTickGet());
00044
00045      DEV_DelayMs(500);
00046
00047      (void)MDS_TaskDelay(50);
00048  }

00055  void TEST_TestTask3(void)
```

```
00056  {
00057      DEV_PutStrToMem((U8*)"\r\nTask3 is running! Tick is: %d",
00058                      MDS_SystemTickGet());
00059
00060      DEV_DelayMs(1000);
00061
00062      (void)MDS_TaskDelay(100);
00063  }
```

编译本节代码，运行结果如下：

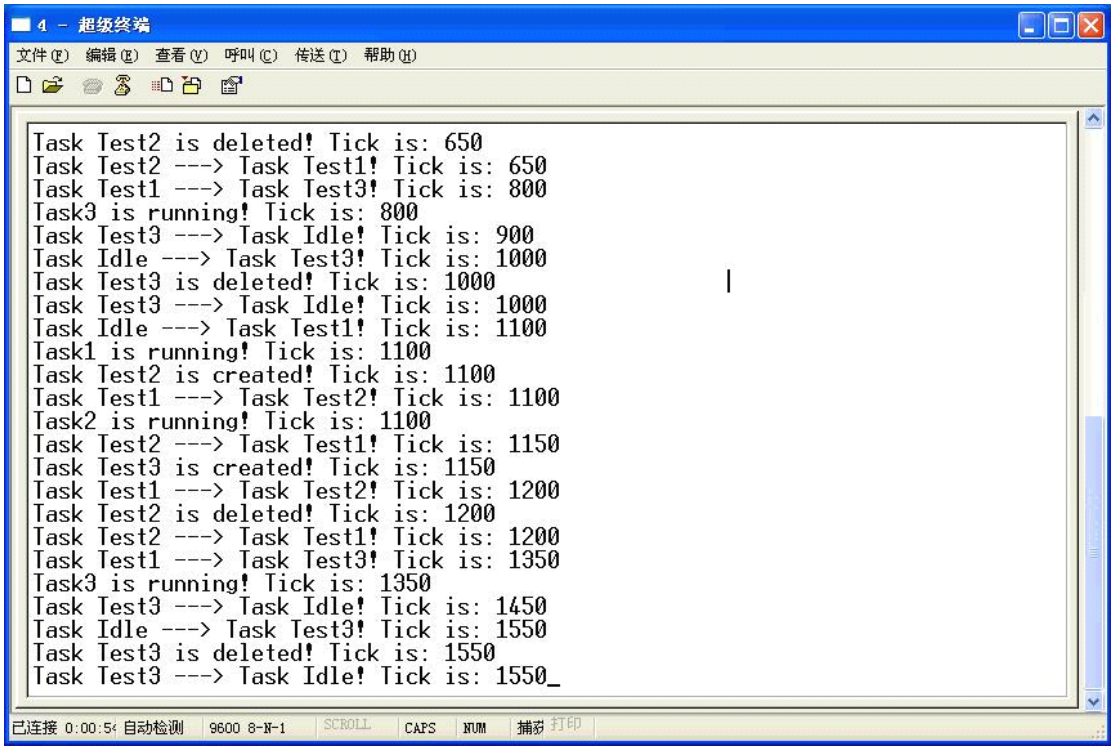


图 75　任务自己申请任务栈的测试

使用工具软件解析本节任务切换过程的数据，结果如下图所示：

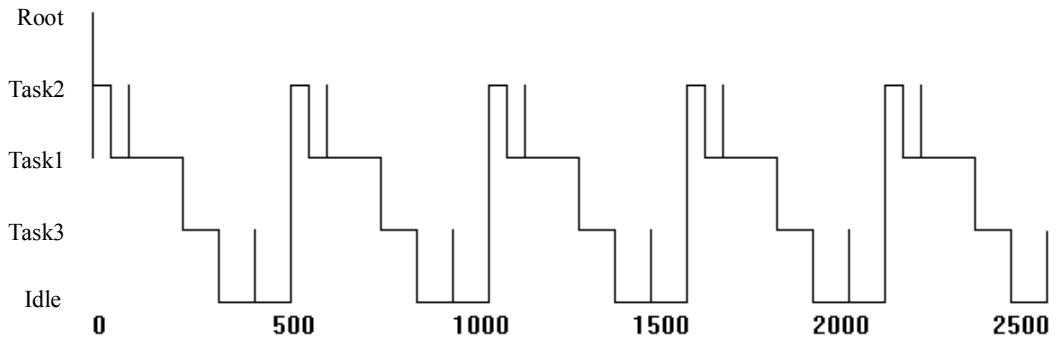


图 76　5.6 节任务切换过程图

第 7 节 任务保护功能

在 Mindows 中我们可以使用 MDS_TaskDelete 函数删除一个任务，无论这个被删除的任务处于何种状态都可以被删除，这样做存在一定的风险，比如说被删除的任务已经申请了一些资源，它在被其它任务删除时还没来得及释放这些资源，那么这些资源就会泄露。比如说被删除的任务被删除时正在对一些数据结构进行操作，此时被删除就会导致这些数据结构不完整，可能会对其它使用这些数据结构的任务产生不良影响。再比如说被删除的任务已经获取了信号量，它在被其它任务删除时还没来得及释放这个信号量，那么这个信号量就可能导致任务死锁。

为了解决这个问题，本节将增加任务保护功能，处于被保护状态下的任务不能被随意删除，只有当这个任务被解除保护状态后才可以被删除。

这个功能很容易实现，我们可以为任务 TCB 中的 uiTaskFlag 再增加一个标志，当任务被保护时就将这个标志置位，当任务解除保护状态时就将这个标志清 0，同时为了使任务保护功能能够嵌套，还需要在 TCB 中新增加一个 uiTaskSafeCnt 变量用来记录任务被保护的次数，任务被保护一次该变量值加 1，解除保护一次该变量值减 1，该变量在创建任务时需要初始化为 0。

下面是任务保护函数 MDS_TaskSafe 和解除任务保护函数 MDS_TaskUnsafe 的源代码，非常好理解，不再详细介绍。

```
00555  U32 MDS_TaskSafe(void)
00556  {
00557      (void)MDS_IntLock();
00558
00559      /* 计数值为空 */
00560      if(TASKSAFEEMPTY == gpstrCurTcb->uiTaskSafeCnt)
00561      {
00562          /* 置任务为保护状态 */
00563          gpstrCurTcb->uiTaskFlag |= TASKSAFEFLAG;
00564
00565          /* 增加保护计数值 */
00566          gpstrCurTcb->uiTaskSafeCnt++;
00567      }
00568      /* 计数值为满，返回失败 */
00569      else if(TASKSAFEFULL == gpstrCurTcb->uiTaskSafeCnt)
00570      {
00571          (void)MDS_IntUnlock();
00572
00573          return RTN_FAIL;
00574      }
00575      else
00576      {
00577          /* 增加保护计数值 */
00578          gpstrCurTcb->uiTaskSafeCnt++;
00579      }
00580
00581      (void)MDS_IntUnlock();
00582
00583      return RTN_SUCD;
00584  }
```

```
00593  U32 MDS_TaskUnsafe(void)
00594  {
00595      (void)MDS_IntLock();
00596
00597      /* 计数值为 1 */
00598      if(1 == gpstrCurTcb->uiTaskSafeCnt)
00599      {
00600          /* 置任务为未保护状态 */
00601          gpstrCurTcb->uiTaskFlag &= (~((U32)TASKSAFEFLAG));
00602
00603          /* 减少保护计数值 */
00604          gpstrCurTcb->uiTaskSafeCnt--;
00605      }
00606      /* 计数值为空, 返回失败 */
00607      else if(TASKSAFEEMPTY == gpstrCurTcb->uiTaskSafeCnt)
00608      {
00609          (void)MDS_IntUnlock();
00610
00611          return RTN_FAIL;
00612      }
00613      else
00614      {
00615          /* 减少保护计数值 */
00616          gpstrCurTcb->uiTaskSafeCnt--;
00617      }
00618
00619      (void)MDS_IntUnlock();
00620
00621      return RTN_SUCD;
00622  }
```

在使用这两个函数时要保证任务运行的保护操作与解保护操作次数相等。

除了直接调用 MDS_TaskSafe 和 MDS_TaskUnsafe 函数可以实现对任务的保护控制之外，在互斥信号量中也提供了这个功能，当互斥信号量在创建时的选项里指定了该功能，那么当一个任务第一次获取到该信号量时就会被自动保护起来，无法被删除，直至该任务最后一次完全释放了该信号量后该任务才可被删除。

这个功能实现起来也很简单，在 MDS_SemTake 函数里如果发现该信号量具有任务保护功能，并且是互斥信号量，那么在任务第一次获取该信号量时 MDS_SemTake 函数就会调用 MDS_TaskSafe 函数将当前任务保护起来。在 MDS_SemGive 函数里如果发现该信号量具有任务保护功能，并且是互斥信号量，那么在任务最后一次释放该信号量时 MDS_SemGive 函数就会调用 MDS_TaskUnsafe 函数解除当前任务的保护状态。细节不再介绍，请读者自行阅读代码。

下面我们构造 3 个测试任务来测试一下任务保护功能，其中 TEST_TestTask1 任务开始时使用 MDS_TaskSafe 函数将自己保护起来，当系统时间大于 20 秒后再使用 MDS_TaskUnsafe 函数解保护。TEST_TestTask2 任务开始时获取一个信号量 gpstrSemMut，该信号量为具有保护功能的互斥信号量，当系统时间大于 30 秒后再释放该信号量。

```
gpstrSemMut = MDS_SemCreate((M_SEM*)NULL, SEMSAFE | SEMMUT | SEMPRIO, SEMFULL);
```

TEST_TestTask3 任务一直在尝试删除 TEST_TestTask1 和 TEST_TestTask2 任务，这 3 个任务的代码如下：

```
00018  void TEST_TestTask1(void)
00019  {
00020      /* 保护任务 */
00021      (void)MDS_TaskSafe();
00022
00023      DEV_PutStrToMem((U8*)"\r\nTask1 is protected! Tick is: %d",
00024                      MDS_SystemTickGet());
00025
00026      while(1)
00027      {
00028          DEV_PutStrToMem((U8*)"\r\nTask1 is running! Tick is: %d",
00029                          MDS_SystemTickGet());
00030
00031          DEV_DelayMs(1000);
00032
00033          (void)MDS_TaskDelay(100);
00034
00035          /* 20秒后, 该任务解保护, 可以被删除 */
00036          if(MDS_SystemTickGet() > 2000)
00037          {
00038              DEV_PutStrToMem((U8*)"\r\nTask1 will be unprotected! Tick is: %d",
00039                              MDS_SystemTickGet());
00040
00041              (void)MDS_TaskUnsafe();
00042
00043              (void)MDS_TaskDelay(DELAYWAITFEV);
00044          }
00045      }
00046  }

00053  void TEST_TestTask2(void)
00054  {
00055      /* 获取互斥信号量 */
00056      if(RTN_SUCD == MDS_SemTake(gpstrSemMut, SEMWAITFEV))
00057      {
00058          DEV_PutStrToMem((U8*)"\r\nTask2 is protected! Tick is: %d",
00059                          MDS_SystemTickGet());
00060      }
00061
00062      while(1)
00063      {
00064          DEV_PutStrToMem((U8*)"\r\nTask2 is running! Tick is: %d",
00065                          MDS_SystemTickGet());
00066
00067          DEV_DelayMs(1000);
00068
00069          (void)MDS_TaskDelay(100);
00070
00071          /* 30秒后, 该任务释放信号量, 解保护, 可以被删除 */
00072          if(MDS_SystemTickGet() > 3000)
00073          {
00074              DEV_PutStrToMem((U8*)"\r\nTask2 will be unprotected! Tick is: %d",
00075                              MDS_SystemTickGet());
00076
00077              (void)MDS_SemGive(gpstrSemMut);
00078
00079              (void)MDS_TaskDelay(DELAYWAITFEV);
00080          }
00081      }
```

```
00082  }

00089  void TEST_TestTask3(void)
00090  {
00091      while(1)
00092      {
00093          DEV_PutStrToMem((U8*)"\r\nTask3 is running! Tick is: %d",
00094                          MDS_SystemTickGet());
00095
00096          (void)MDS_TaskDelay(100);
00097
00098          /* 任务1没被删除时删除任务1 */
00099          if(NULL != gpstrTask1)
00100          {
00101              if(RTN_SUCD == MDS_TaskDelete(gpstrTask1))
00102              {
00103                  gpstrTask1 = (M_TCB*)NULL;
00104
00105                  DEV_PutStrToMem((U8*)"\r\nTask1 is deleted! Tick is: %d",
00106                                  MDS_SystemTickGet());
00107              }
00108              else
00109              {
00110                  DEV_PutStrToMem((U8*)"\r\nTask1 can't be deleted! Tick is: %d",
00111                                  MDS_SystemTickGet());
00112              }
00113          }
00114
00115          /* 任务2没被删除时删除任务1 */
00116          if(NULL != gpstrTask2)
00117          {
00118              if(RTN_SUCD == MDS_TaskDelete(gpstrTask2))
00119              {
00120                  gpstrTask2 = (M_TCB*)NULL;
00121
00122                  DEV_PutStrToMem((U8*)"\r\nTask2 is deleted! Tick is: %d",
00123                                  MDS_SystemTickGet());
00124              }
00125              else
00126              {
00127                  DEV_PutStrToMem((U8*)"\r\nTask2 can't be deleted! Tick is: %d",
00128                                  MDS_SystemTickGet());
00129              }
00130          }
00131      }
00132  }
```

如果任务保护功能有效的话，我们应该看到在前 20 秒内，TEST_TestTask1 任务一直无法被删除，20 秒后 TEST_TestTask1 任务才被删除，在前 30 秒内，TEST_TestTask2 任务一直无法被删除，30 秒后 TEST_TestTask2 任务才被删除。

编译本节代码，部分运行结果如下图所示，更多数据和视频请到我的博客 blog.sina.com.cn/ifreecoding 下载。

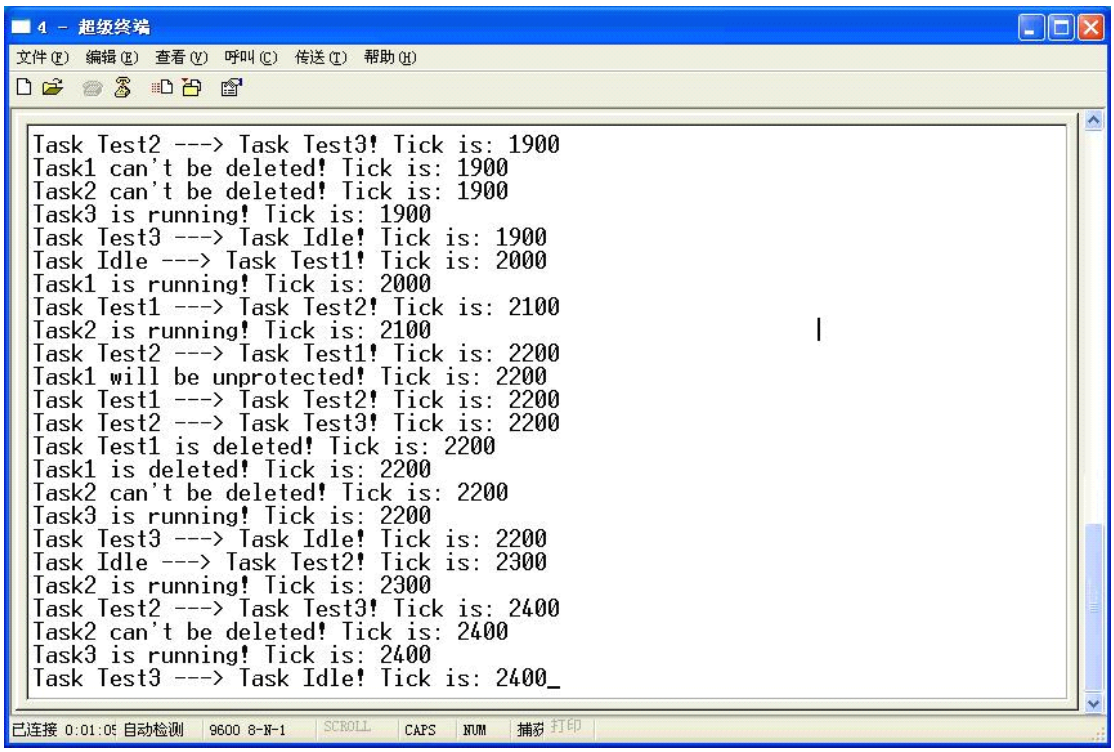


图 77　任务保护测试

使用工具软件解析本节任务切换过程的数据，结果如下图所示，可以看到 TEST_TestTask1 任务在 20 秒前一直在运行，在 20 秒后被删除了，TEST_TestTask2 任务任务在 30 秒前一直在运行，在 30 秒后被删除了，与我们的设计是一致的。

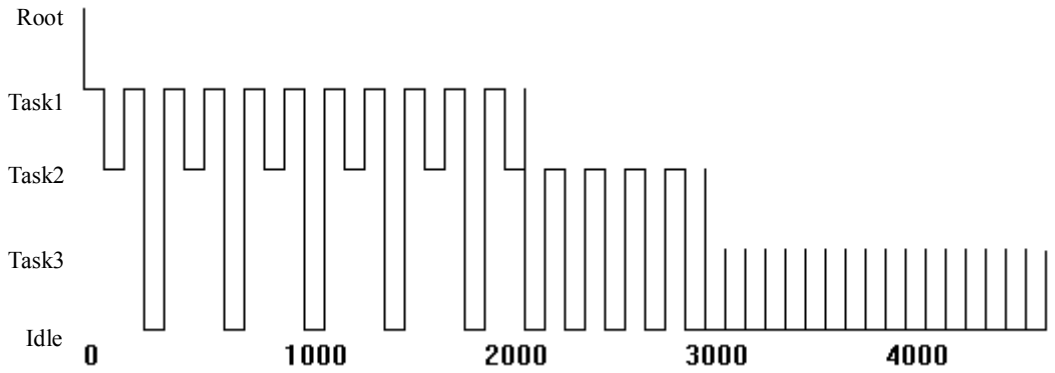


图 78　5.7 节任务切换过程图

附录 1　Wanlix 接口函数

接口函数列表：

```
W_TCB* WLX_TaskCreate(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize)
void WLX_TaskSwitch(W_TCB* pstrTcb)
```

函数说明：

◆

```
W_TCB* WLX_TaskCreate(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize)
```

函数描述：
创建一个任务。

入口参数：
vfFuncPointer：创建任务所使用函数的指针。
pucTaskStack：任务所使用堆栈的最低起始地址。
uiStackSize：堆栈大小, 单位: 字节。

返回值：
NULL：创建任务失败。
其它：任务的 TCB 指针。

◆

```
void WLX_TaskSwitch(W_TCB* pstrTcb)
```

函数描述：
调用该函数将发生任务切换，切换到入口参数 TCB 的任务。

入口参数：
pstrTcb：即将运行的任务的 TCB 指针。

其它说明：
不能在中断中调用该函数。

附录 2　参考资料

1. ADuC7019_7020_7021_7022_7024_7025_7026_7027_7028_7029
2. ARM Architecture Reference Manual
3. Procedure Call Standard for the ARM Architecture
4. CM3 权威指南中文版（www.ouravr.com 热心网友校对版）
5. Cortex-M3 Technical Reference Manual (TRM) (Cortex-M3 技术参考手册)
6. ARMv7-M Architecture Application Level Reference Manual
7. Stellaris® Peripheral Driver Library USER'S GUIDE
8. EK-LM3S8962 Firmware Development Package USER'S GUIDE
9. Stellaris® LM3S8962 Evaluation Board USER'S MANUAL
10. LM3S8962 Microcontroller DATA SHEET
11. STM32F10x 中文参考手册
12. DBLEC-STM32A 开发板用户手册

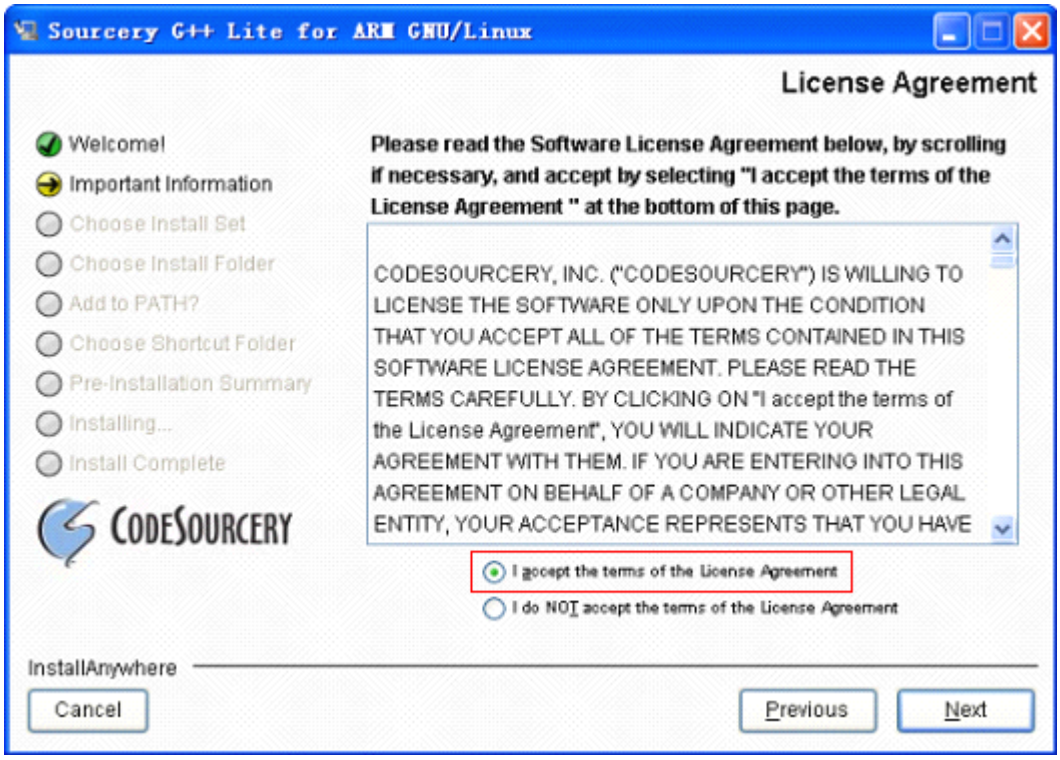
附录 3　Wanlix 开发环境安装

以下介绍均是在 Xp 系统下

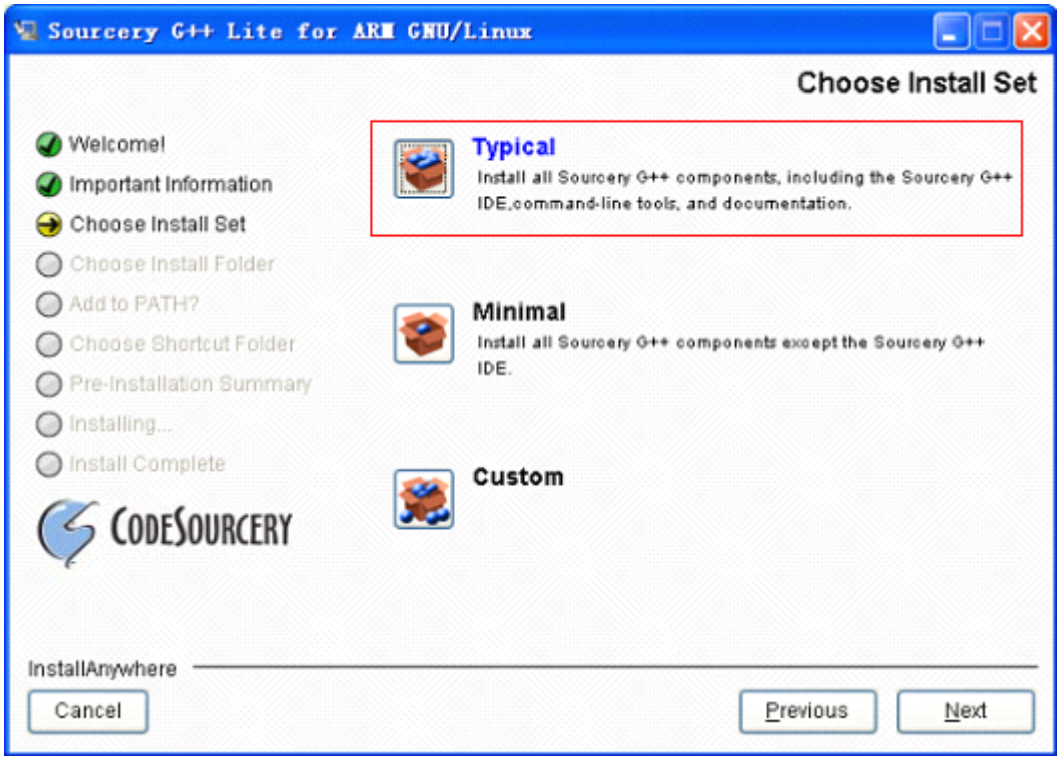
1、安装 GNU 工具链

我使用的是 CodeSourcery 的 GNU ARM 工具链编译的 Wanlix，因此需要安装 GNU 工具链，可以在网站 http://blog.sina.com.cn/ifreecoding 的“资源下载”中获取，或者在其它网站下载。

在下面的界面选择红框内的“I accept the terms of License Agreement”，然后点“Next”按钮。



在下面的界面选择红框内的“Typical”，然后点“Next”按钮。



其余所有界面全部默认，点“Next”直至完成安装。

需要注意的是，最好不要改安装路径，否则在我的源代码可能因为路径不通编译不通过。

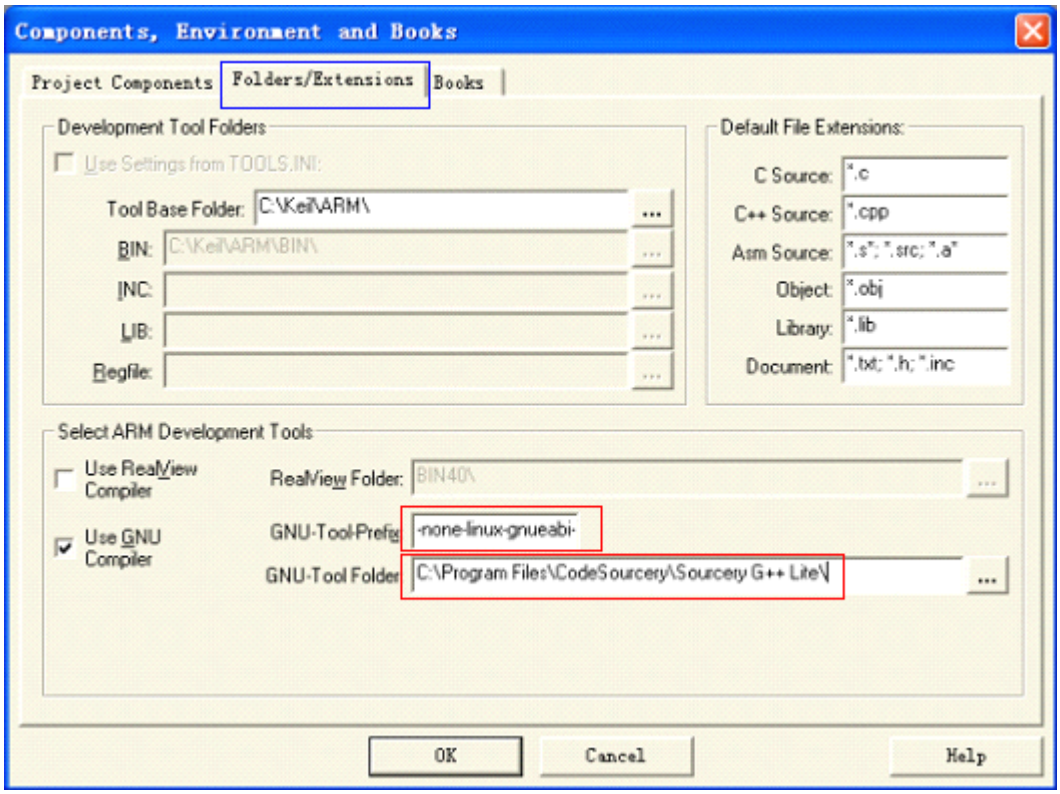
2、安装 Keil MDK4.20

我使用的是 Keil MDK4.20 版本作为开发环境，可以在网站的“资源下载”中获取，或者在其它网站下载。

安装时不要改任何默认选项，尤其是安装路径，否则在我的源代码可能因为路径不通编译不通过，该填信息的地方随便填，一路 Next 直至完成安装。

安装后，打开 Keil，在上方工具栏里有一个“品”字型的图标，如下图，点开。

便会打开下面的对话框，点上面蓝色地方中间的标签，然后将第一个红框内改为“arm-none-linux-gnueabi-”，将第二个红框内改为“C:\Program Files\CodeSourcery\Sourcery G++ Lite\”。



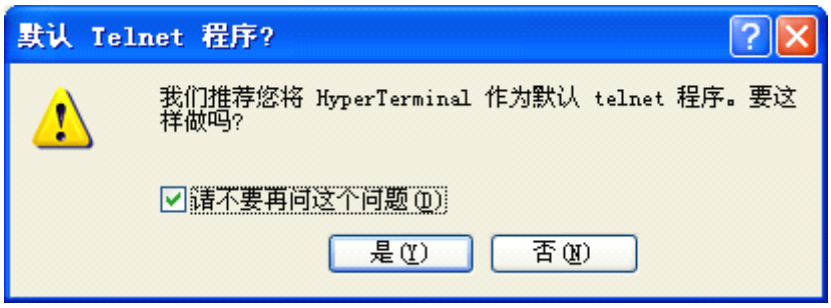
3、串口设置

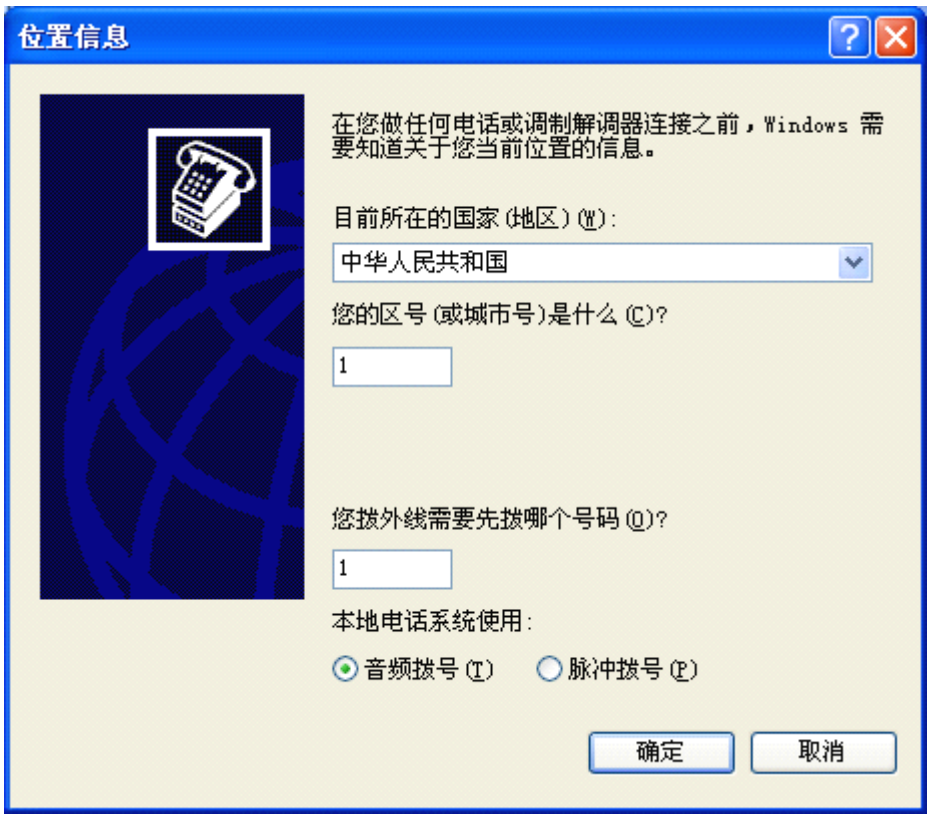
通过上面 2 步就可以编译 Wanlix 了，如果你有开发板，可以通过串口观察 Wanlix 的输出。

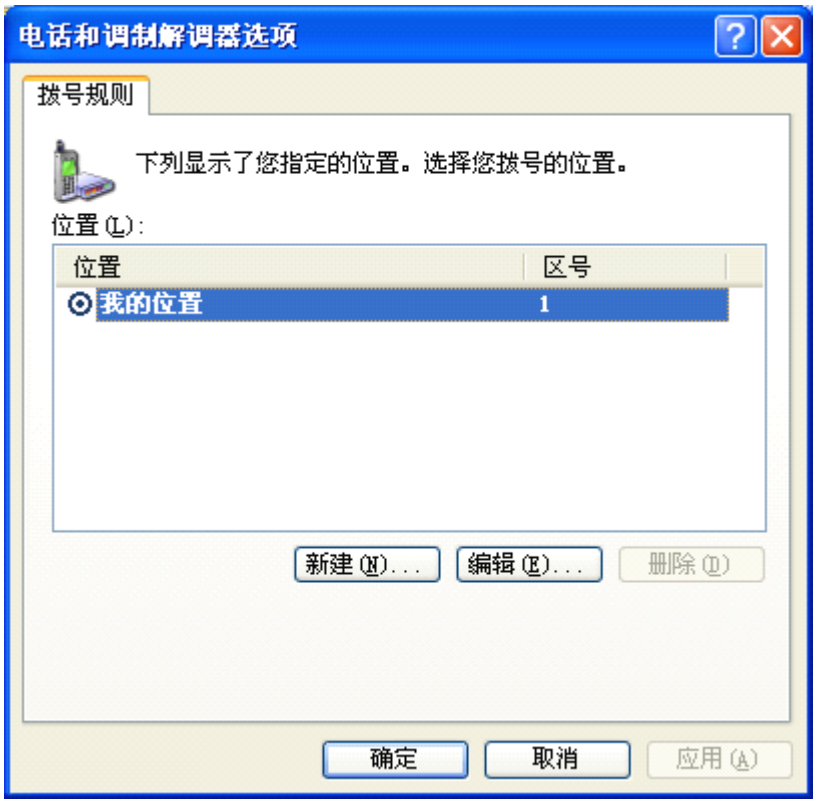
我使用的是 windows 自带的串口工具——hypertrm，如果你的 windows 里找不到该程序，可以在网站的“资源下载”中获取，或者在其它网站下载，或者下载其它串口工具。

以 hypertrm 为例介绍一下串口设置，不同的串口工具大同小异。

第一次使用 hypertrm 时需要配置，按下面图处理：

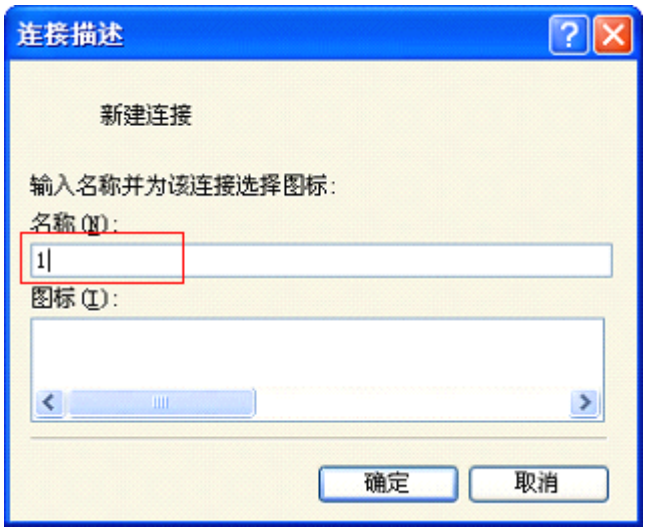




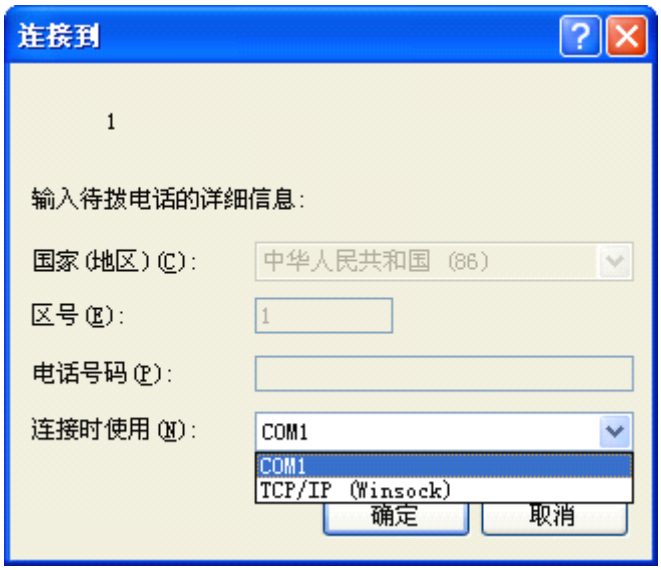


每次打开都会需要进行下面的设置。

下图红框内名称可以随便起。



下图需要选择使用的计算机串口，开发板接到哪个串口上了就选择哪个。



下图设置串口属性，只需要将第一个红框内改为“9600”即可，其它保持不变。我使用的都是 9600 波特率。


COM1 属性
端口设置
每秒位数(B): 9600
数据位(D): 8
奇偶校验(P): 无
停止位(S): 1
数据流控制(F): 硬件
还原为默认值(R)
确定
取消
应用(A)

附录 4　Mindows 开发环境安装

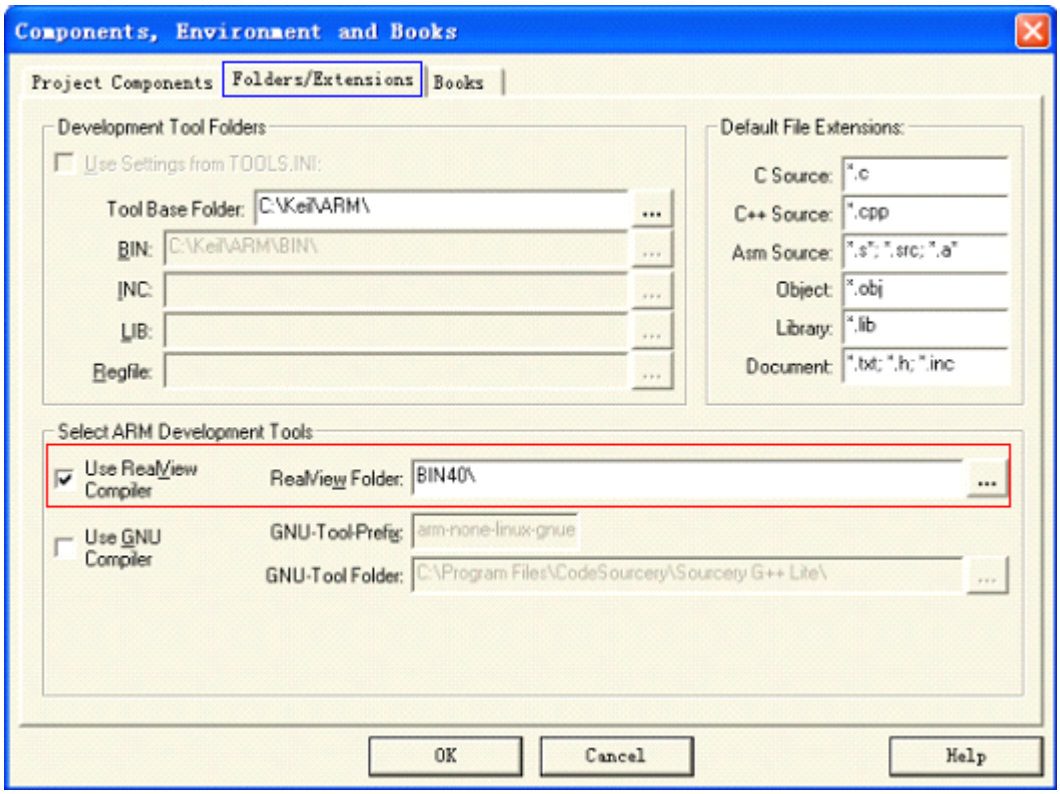
以下介绍均是在 **Xp** 系统下

1、更改 Keil 编译工具链

Keil 的安装请参考附录 3。在第 3 和第 4 章中，我们使用 GNU 工具链来编译代码，在第 5 章，我们使用 Keil 自带的 RealVeiw 工具链编译代码，在 Keil 中更换工具链的方式如下：

打开 Keil，在上方工具栏里有一个“品”字型的图标，如下图，点开。

便会打开下面的对话框，点上面蓝色地方中间的标签，然后选中红框的复选框，将红框内改为“BIN40\”。



2、安装 LM3S8962 单板开发包

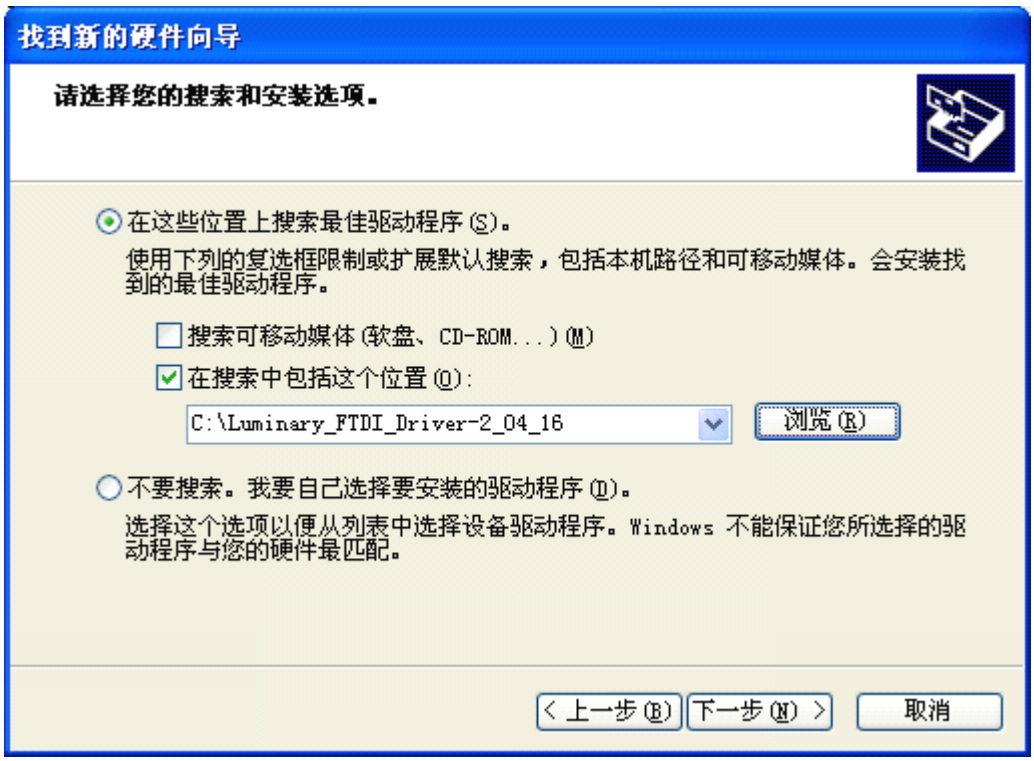
请在 http://blog.sina.com.cn/ifreecoding 网站的“资源下载”中下载 LM3S8962 开发包的安装文件——SW-LM3S-7611.rar，解压后进行安装，请安装到默认路径 C:\StellarisWare。

3、安装 LM3S8962 单板 USB 驱动

LM3S8962 单板 USB 接口集成了供电、仿真、烧片和串口通信的功能，需要安装驱动。请在 http://blog.sina.com.cn/ifreecoding 网站的“资源下载”中下载 LM3S8962 单板的 USB

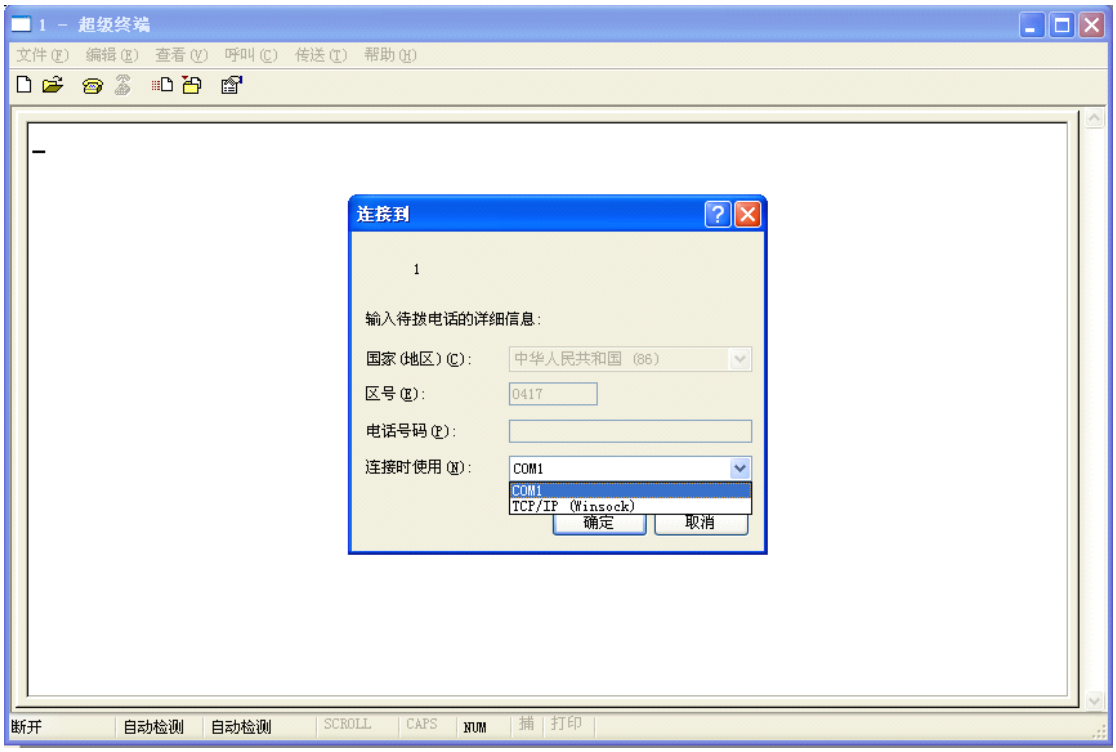
驱动安装文件——Luminary_FTDI_Driver-2_04_16.zip。将其解压到C:\Luminary_FTDI_Driver-2_04_16，然后使用USB连接单板和电脑，会出现硬件安装向导，按下图选择，点击“下一步”。

按下图选择，点击“下一步”。

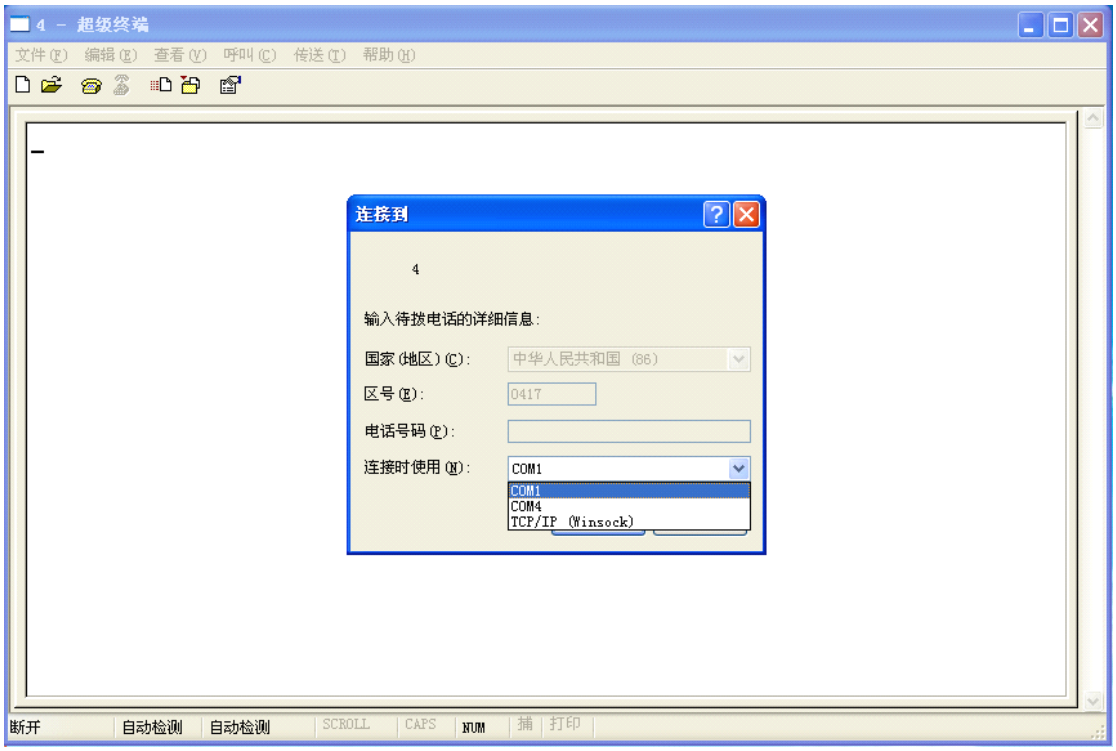


之后会再自动弹出2次“找到新的硬件向导”，重复上面过程即可完成安装。

再将单板与电脑断开，打开串口工具，可以看到我使用的这台电脑只有一个串口 COM1，如下图：



关闭串口工具，将单板连接到电脑上，再重新打开串口工具，会发现多了一个串口 COM4，这个新增加的串口就是 USB 虚拟的串口，选中 COM4 即可与单板进行通信了。

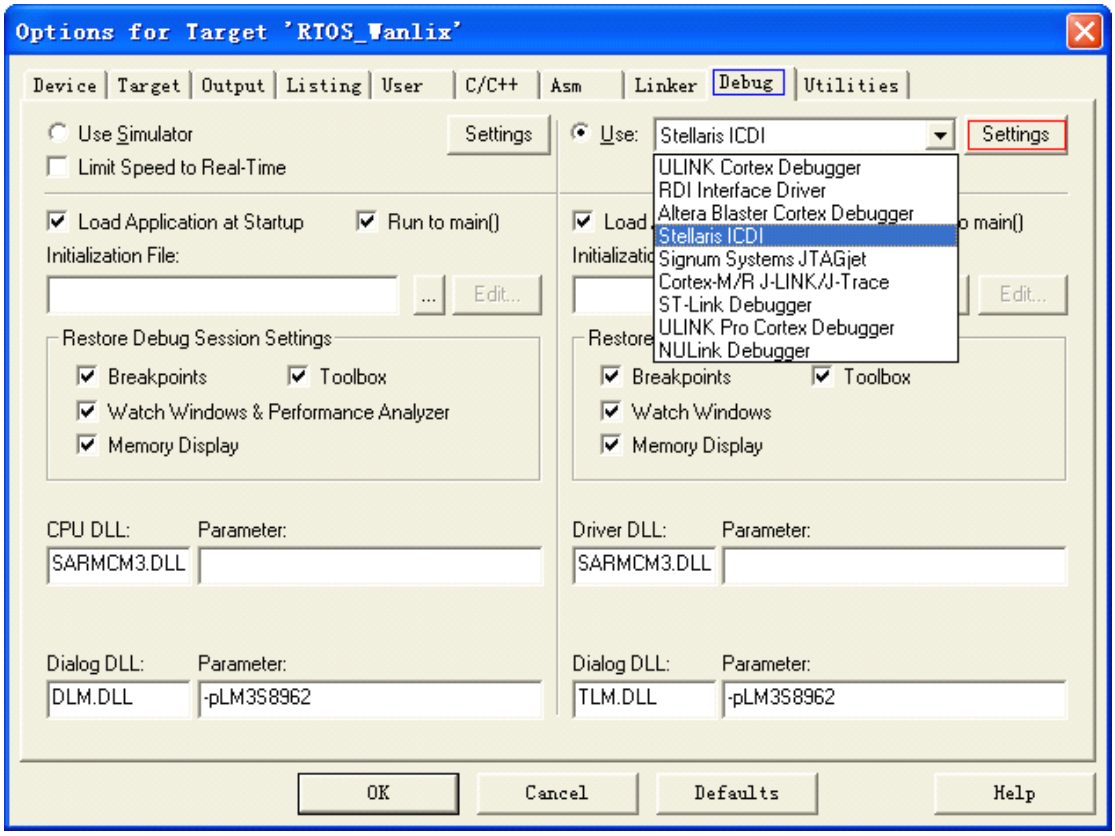


串口的其它设置请参考附录 3。

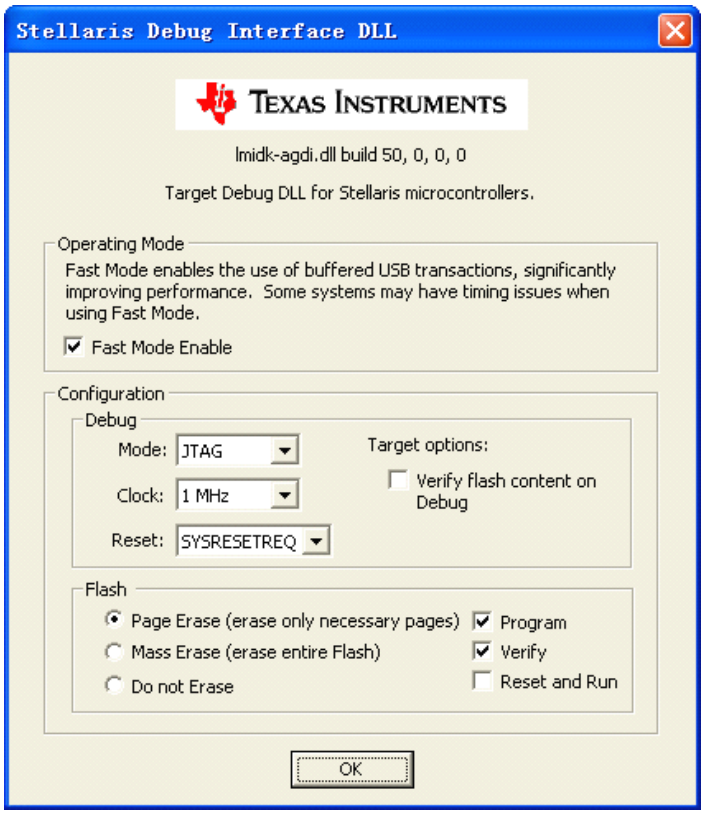
4、LM3S8962 仿真器的设置

打开 Keil，点击工具栏上的图标，在新打开的窗口中选择上面的“debug”标签，如

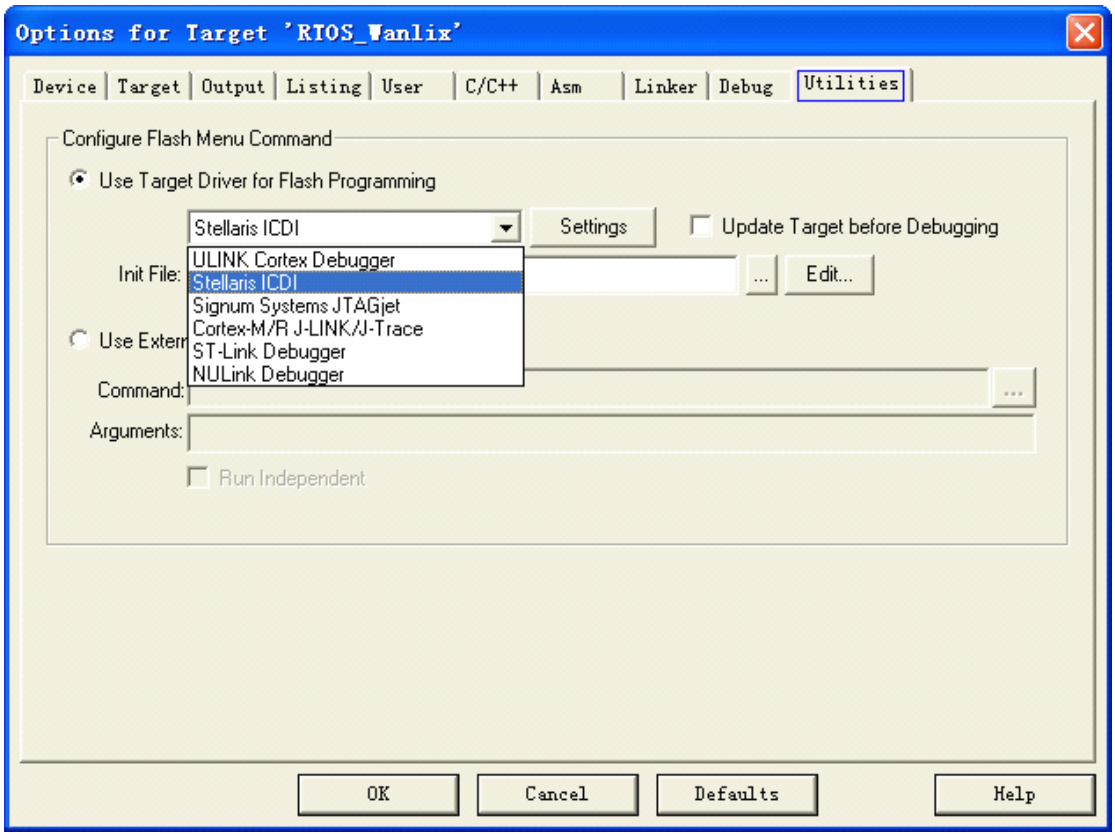
下：



在下拉框中选择“Stellaris ICDI”，然后点击红框的“Setting”，打开下面窗口，并按下面的窗口进行设置：

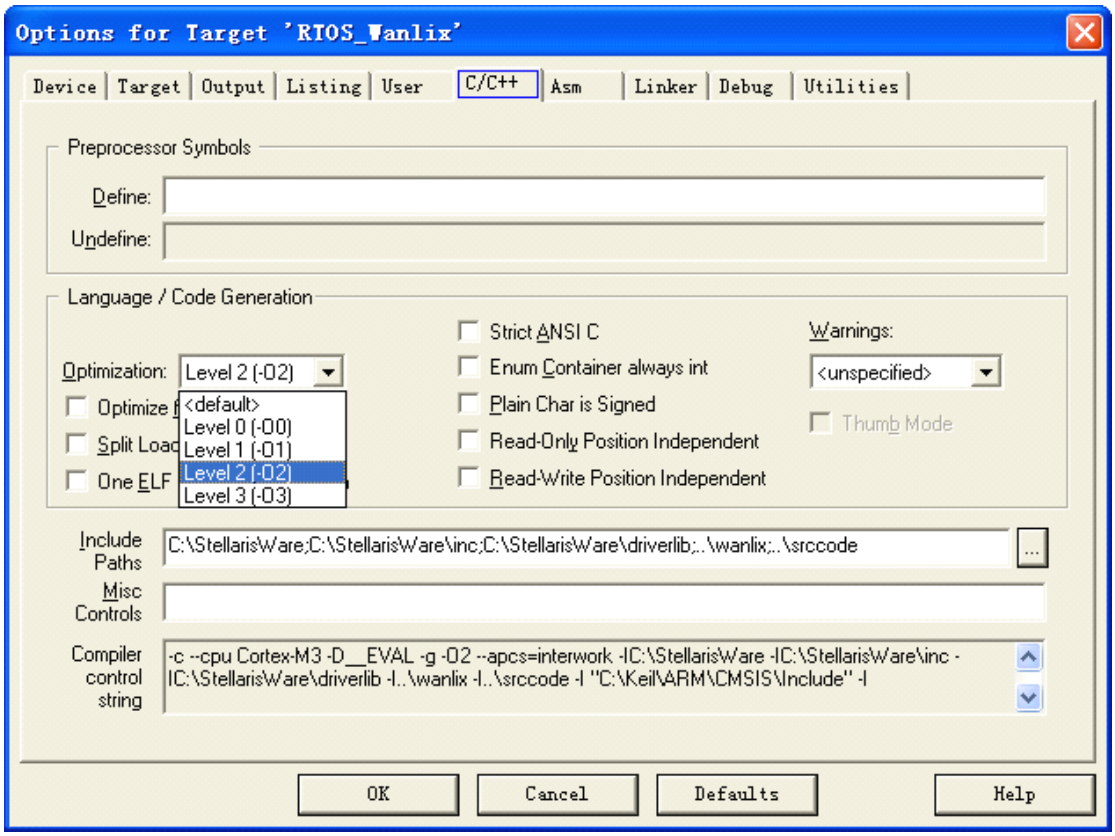


之后再打开“Utilities”标签的页面，同样选择“Stellaris ICDI”完成设置。



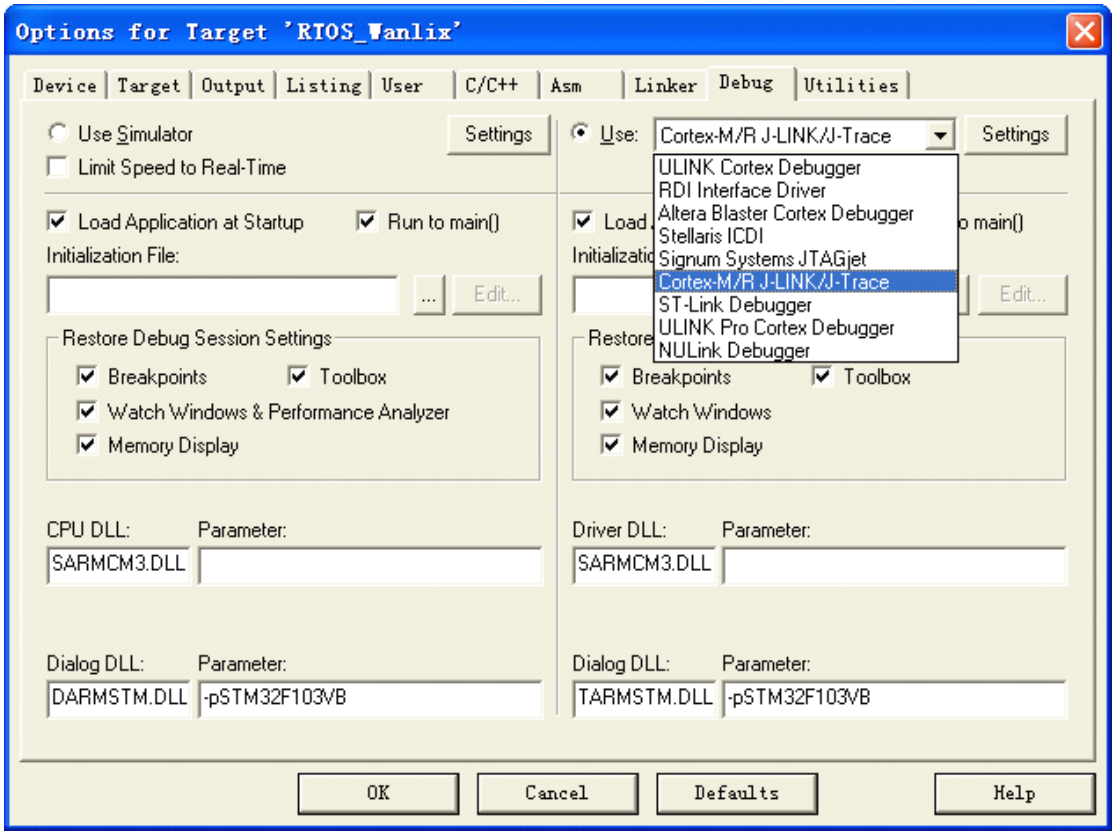
5、LM3S8962 的加载方式

经过上述设置开发环境就已经建好了，打开 Keil，点击这 2 个图标 之一编译代码，然后点击 图标将编译后的代码烧进芯片，此时复位单板就可以看到程序运行的结果了。如果你想使用仿真器调试，点击 图标即可。请记住，在代码修改后，需要先编译，然后再烧写，最后才能仿真，而且在仿真时需要先将编译选项修改为 O0 后再编译，如下图，打开“C/C++”标签，改为 O0 选项。

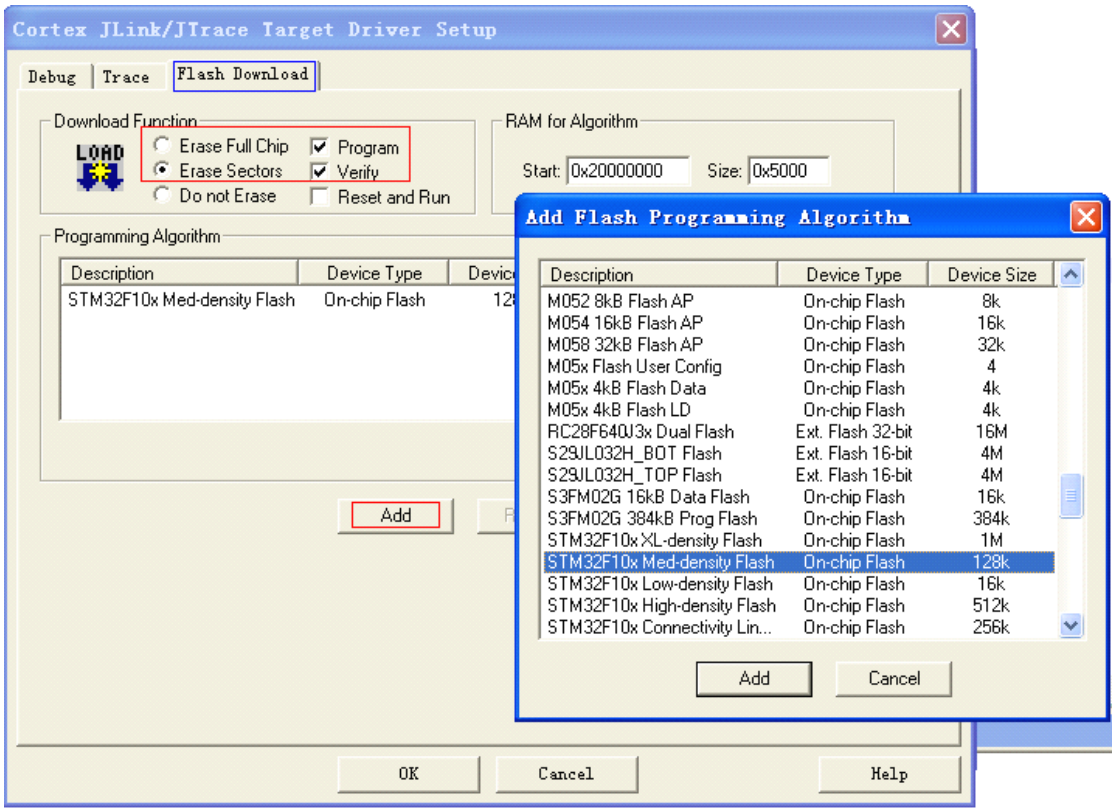


6、STM32F103VB 仿真器的设置

我使用的是 J-LINK 仿真器，如同 LM3S8962 仿真器的设置，需要在“debug”和“Utilities”标签中将仿真器设置为 “Cortex-M/R J-LINK/J-Trace”。



并在“Setting”中选中“Flash Download”标签，点击红框的“Add”按钮，添加板子上芯片的类型，比如说我使用的是中密度的 103 芯片，就选中图中的那款芯片。其余的设置也修改为图中所示：



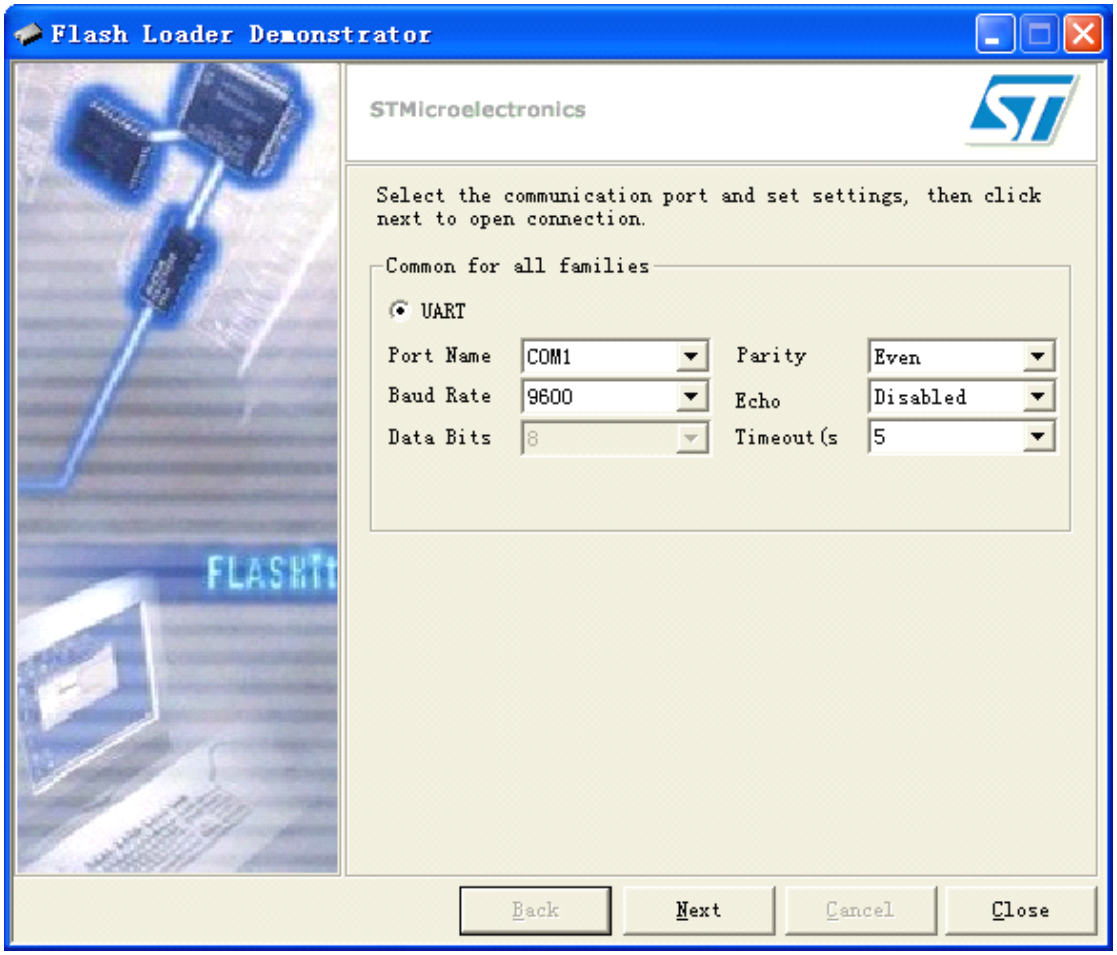
7、STM32F103VB 的加载方式

如果你有仿真器的话，那么 STM32F103VB 的加载方式与 LM3S8962 的加载方式是一样的，请参考上面的介绍。

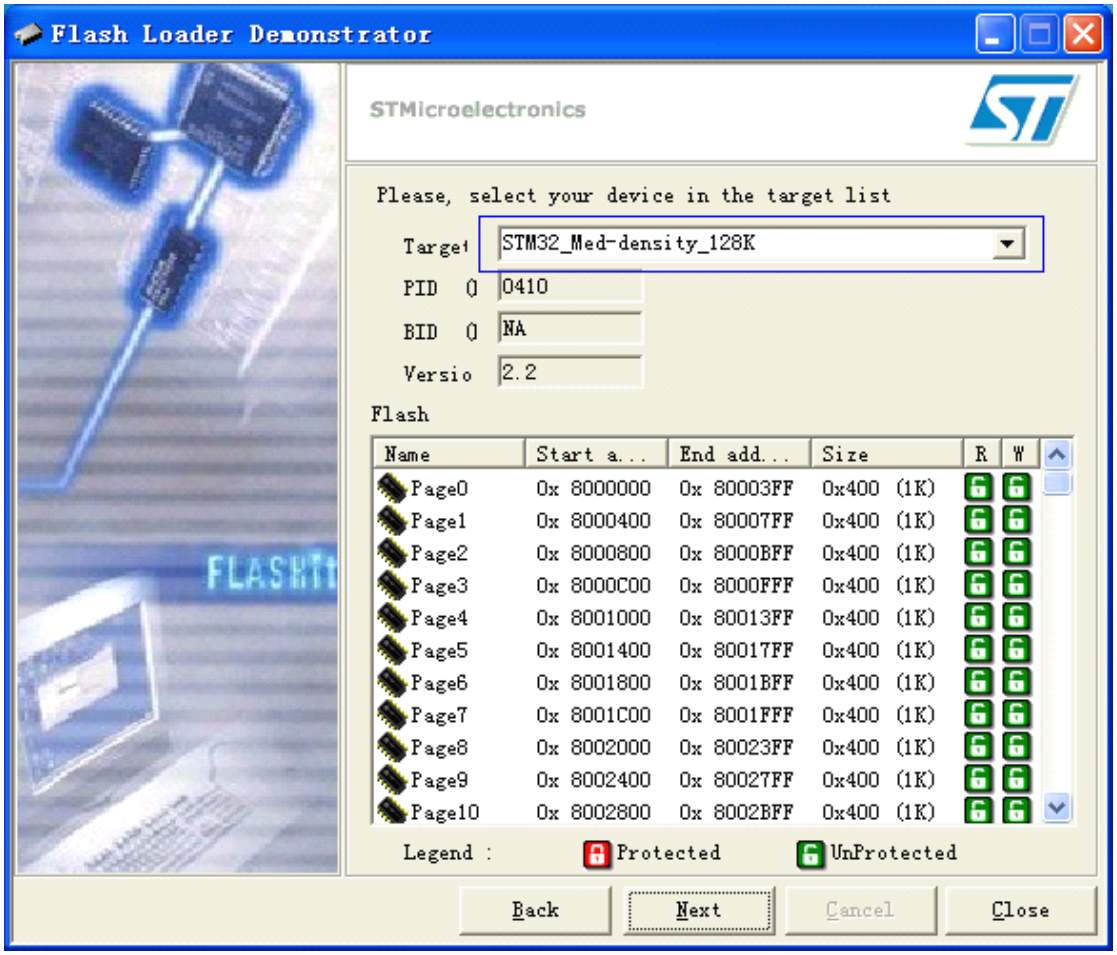
如果你没有仿真器，那么可以通过串口加载。首先在 http://blog.sina.com.cn/ifreecoding 网站的“资源下载”中下载 STM32F103VB 单板的串口编程工具——Flash_Loader_Demonstrator_v2.2.0_Setup.zip，解压后安装。

在断电状态下更改板子的启动模式，在板子的用户手册里可以查找到，设置为 ISP 模式，即 BOOT0=1，BOOT1=0。

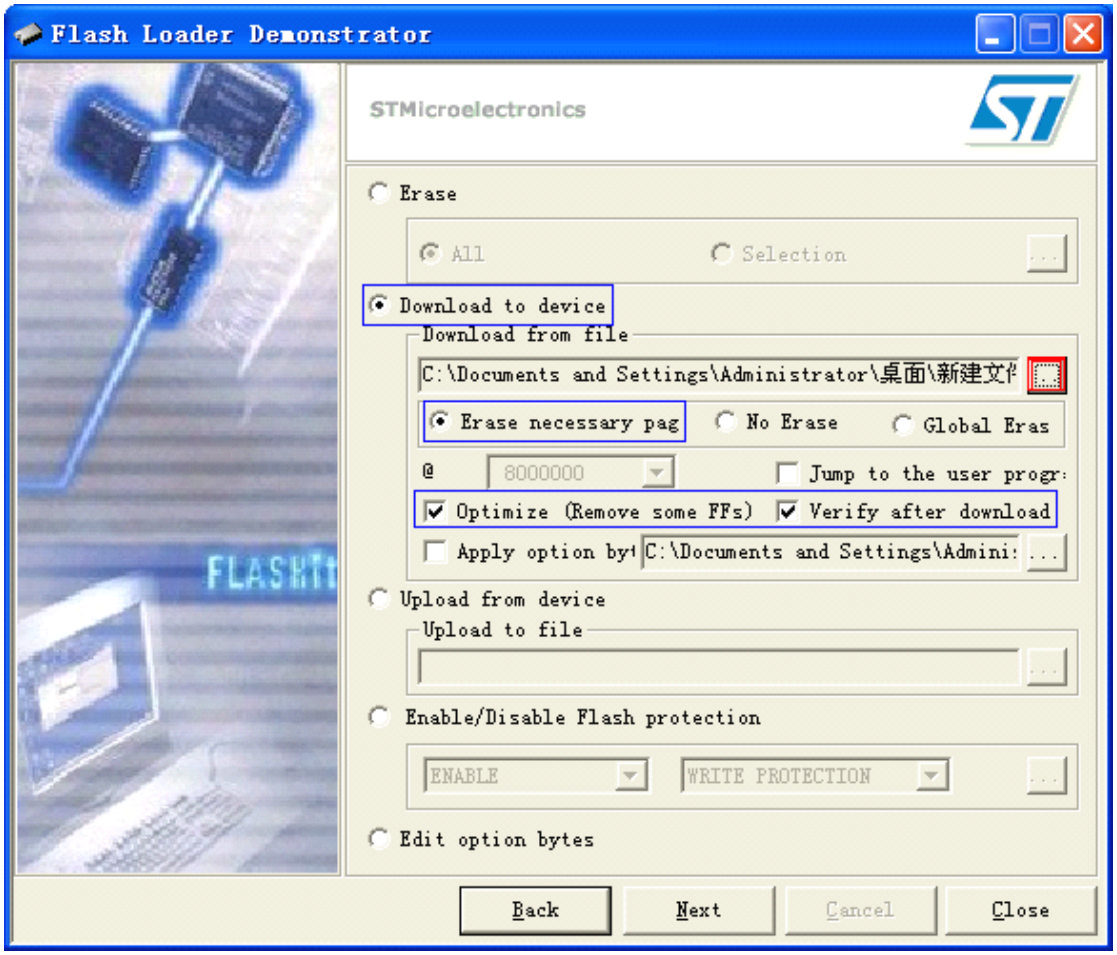
将板子的串口 1 连接到 PC 机的串口上，将单板上电，并在 windows 的开始菜单中运行新安装的的串口加载工具 Flash Loader Demonstrator，界面如下图所示，并按照下图参数进行设置。



点击 Next 进行下一步，



在蓝框内选择对应的芯片容量，点击 Next 进行下一步，



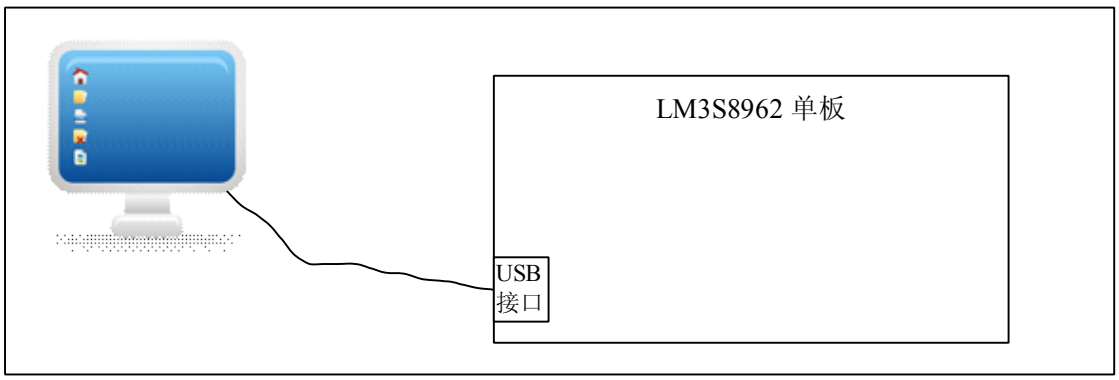
蓝框内按上图设置，点击红框按钮，在源代码所在路径“\RTOS_Wanlix\outfile”下选择 hex 文件，然后回到本窗口，点击 Next 开始烧片，直至烧片、校验成功。

将单板电源断开，查找单板手册，将板子的启动模式更改为从 FLASH 启动，即 BOOT0=0 的模式。

至此通过串口烧片完成，单板再上电时就可以执行新烧入的程序了。

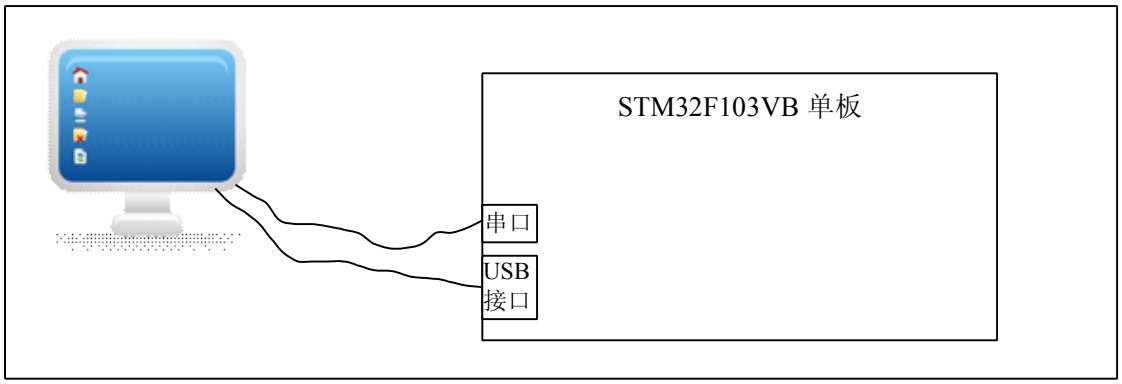
请注意，这个串口烧写工具不是很稳定，如果它连接不上单板请重新启动单板和电脑再重新尝试，如果在烧写、校验过程中出错，请在上图中选中第一项“Erase”，先擦除整个 FLASH 再重复烧写步骤。

8、LM3S8962 单板连接示意图



USB 接口集成了供电、调试、烧片和串口打印数据功能。

9、STM32F103VB 单板连接示意图



需要使用 USB 接口供电，使用串口烧片和打印数据。